Distributed simulation of multi-agent systems and their environments

Wang, Lihua

2005

Wang, L. (2005). Distributed simulation of multi-agent systems and their environments.
Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2521>

<https://doi.org/10.32657/10356/2521>

Nanyang Technological University

Downloaded on 26 Apr 2023 10:03:14 SGT

NANYANG TECHNOLOGICAL UNIVERSITY

DISTRIBUTED SIMULATION OF MULTI-AGENT SYSTEMS AND THEIR ENVIRONMENTS

A thesis submitted to the Nanyang Technological University in fulfillment of the
requirement for the degree of

Master of Engineering

By

WANG LIHUA

School of Computer Engineering

2004

ABSTRACT.....	I
ACKNOWLEDGEMENTS.....	II
TABLE OF CONTENTS.....	III
LIST OF FIGURES.....	VI
LIST OF TABLES.....	IX

ABSTRACT

Distributed simulation enables participants situated in different geographical locations to share a common virtual world, which is called a Distributed Virtual Environment (DVE). Among the different research topics concerned with DVEs, there is a current trend of linking Multi-Agent Systems and DVEs together. With the properties of autonomy, social ability, reactivity and pro-activeness, agents can be used to represent entities in DVEs, where fast and accurate decision-making is a determining factor of the whole environment.

This thesis provides a description of integrating agents into an HLA-based distributed simulation. It addresses two main issues. One is how to construct the sensor of an agent with different interest management schemes. Using the JADE (Java Agent DEvelopment Framework) agent toolkit and the High Level Architecture (HLA) in our prototype, a minesweeping game, we describe and evaluate some alternative implementations. Due to the dynamic characteristics of agents, a problem of overdue information from the environment is discussed, and we propose an enlarged subscription region method to solve this problem.

Another issue addressed by this project is to resolve concurrent interactions, as the shared environment needs to allow agents to interact with the environment in a causally consistent way. There will usually be either mutually exclusive or collaborative interactions in an agent-based distributed simulation. This thesis presents our research in designing a middleware component called Interaction Resolver (IR) to resolve the effect of concurrent interactions and still guarantee the consistency and causality of the system. The ownership management services provided by the HLA are compared with IRs in resolving mutually exclusive interactions in our prototype minesweeping game. Conclusions are drawn based on the experimental results.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, *Dr. Stephen John Turner*, for his guidance and support at all times. I am very grateful for his understanding and patience. He is such a knowledgeable and nice supervisor who helps me a lot in defining the research topics and solving the problems encountered. His serious research attitude and great knowledge do influence all people around him.

I would also like to express my gratitude to Ng-Goh Siew Lai and Ek Ming Hong, technicians of our Parallel and Distributed Computing Center (PDCC). They maintain the best hardware and software environments, and provide me with all the convenience I need.

Many thanks to all my friends, both near and far, thank all of them for the help, advice, suggestions and discussions; thank all of them for the talks, smiles, support and understanding. It is them that always encourage me to persist in my dreams.

Last but not least, I thank my dear parents, who give me continued love and support unconditionally all these years.

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	Background	1
1.1.1	Agents	1
1.1.2	Distributed Virtual Environments	2
1.2	Motivation	3
1.3	Objectives	4
1.4	Organization of the Thesis	5
2	RELATED WORK	7
2.1	Analytic Simulations versus Virtual Environments	8
2.2	Structure of the Environment	9
2.2.1	System architecture	9
2.2.2	Partitioning the virtual environment	14
2.3	High Level Architecture	16
2.3.1	An Overview of the HLA	16
2.3.2	Ownership Management in the HLA	21
2.3.3	Time Management in the HLA	24
2.4	Approaches to Interest Management	26
2.4.1	Introduction	26
2.4.2	Interest management mechanisms in the HLA	28
2.4.3	Comparison of different interest management schemes	31
2.5	Agents and Simulations	34
3	PROTOTYPE SYSTEM	37
3.1	Purpose of the Prototype System	37
3.2	Description of the Minesweeping Game	38
3.3	Overall Architecture	42
3.4	Object Model - Soldiers and Environment	45
3.5	The Simulation Cycle	50

3.6	Lessons Learnt	53
3.7	Discussion of the Prototype	54
3.7.1	Federation management	54
3.7.2	Declaration and Object management	55
3.7.3	Ownership management	55
3.7.4	Time management	56
3.7.5	Data distribution management	56
3.7.6	Sensor and effector	57
3.7.7	Mutual exclusion	58
4	CONSTRUCTING AGENT SENSORS WITH INTEREST MANAGEMENT	59
4.1	Introduction	59
4.2	Implementations	60
4.3	Dynamic Regions: Problems and Solutions	65
4.3.1	Enlarged subscription region	66
4.3.2	Using advisories	67
4.3.3	Causality problems of advisories	68
4.5	Experimental Results	72
5	RESOLVING CONCURRENT INTERACTIONS .	73
5.1	Introduction	73
5.1.1	Mutually exclusive interactions and collaborative interactions	74
5.1.2	Processing concurrent interactions	75
5.2	Mutually Exclusive Interactions	76
5.2.1	Problems of mutually exclusive interactions in the prototype	76
5.2.2	Solution 1: Ownership Management (OM)	77
5.2.3	Solution 2: Interaction Resolvers (IRs)	80
5.3	Comparison of Solutions: Picking Up the Same Mine	86
5.3.1	Using OM	87
5.3.2	Using IRs	89
5.3.3	Experimental results	91
5.4	Comparison of Solutions: Stepping to the Same Grid Cell	94
5.4.1	Using OM	95
5.4.2	Using IRs	97
5.4.3	Experimental results	99
5.5	Summary	103

6	CONCLUSIONS AND FUTURE WORK.....	105
6.1	Conclusions.....	105
6.2	Future Work.....	106
6.3	Summary.....	108
	APPENDIX A SYSTEM CONFIGURATION.....	109
	APPENDIX B PUBLICATIONS.....	111
	REFERENCES.....	112

LIST OF FIGURES

<i>Figure 2.1: A centralized server architecture</i>	<i>10</i>
<i>Figure 2.2: A distributed serverless architecture.....</i>	<i>11</i>
<i>Figure 2.3: The hybrid architecture</i>	<i>13</i>
<i>Figure 2.4: Cell-based partition.....</i>	<i>14</i>
<i>Figure 2.5: Non-cell-based partition.....</i>	<i>15</i>
<i>Figure 2.6: Software components of an HLA federation.....</i>	<i>17</i>
<i>Figure 2.7: Interfaces between RTI and federates</i>	<i>18</i>
<i>Figure 2.8: The HLA federation development and execution process model.....</i>	<i>20</i>
<i>Figure 2.9 : Diagram of negotiated push</i>	<i>22</i>
<i>Figure 2.10: Diagram of obtrusive pull.....</i>	<i>23</i>
<i>Figure 2.11: Diagram of unobtrusive pull.....</i>	<i>23</i>
<i>Figure 2.12: Two-dimensional routing space example</i>	<i>29</i>
<i>Figure 2.13: Two-dimensional routing space using the region-based approach.....</i>	<i>30</i>
<i>Figure 2.14: Two-dimensional routing space using the grid-based approach</i>	<i>31</i>
<i>Figure 2.15: Irrelevant data is delivered from F2 to F1 even though S and U do not intersect.....</i>	<i>32</i>
<i>Figure 3.1: The environment for the prototype</i>	<i>38</i>
<i>Figure 3.2: Soldier agent architecture of the prototype system</i>	<i>39</i>
<i>Figure 3.3: Logical structure of a simulation cycle of an agent</i>	<i>40</i>
<i>Figure 3.4: Sensor region.....</i>	<i>40</i>
<i>Figure 3.5: The environment with coordinates</i>	<i>41</i>
<i>Figure 3.6: The GUI of the minesweeping game.....</i>	<i>41</i>
<i>Figure 3.7: The initial architecture of the minesweeping game.....</i>	<i>43</i>
<i>Figure 3.8: The final architecture of the minesweeping game.....</i>	<i>43</i>
<i>Figure 3.9: Middleware composed of JADE and the gateway federates</i>	<i>44</i>
<i>Figure 3.10: Object classes for the minesweeping game</i>	<i>46</i>
<i>Figure 3.11: Interaction classes for the minesweeping game</i>	<i>46</i>

<i>Figure 3.12: Illustration of the parameters in the interaction class CommunicationE</i>	47
<i>Figure 3.13: The FED file for the minesweeping prototype</i>	48
<i>Figure 3.14: The simulation cycle of the prototype system</i>	50
<i>Figure 3.15: Flow charts of the simulation</i>	52
<i>Figure 4.1. The object classes of version 2</i>	61
<i>Figure 4.2: The FED file for version 2</i>	62
<i>Figure 4.3: The simulation cycle of version 2</i>	64
<i>Figure 4.4: Enlarged subscription region of the soldier federate</i>	66
<i>Figure 4.5: The simulation cycle of version 2.1</i>	67
<i>Figure 4.6: The simulation cycle of version 2.2</i>	68
<i>Figure 4.7: Possible error when using advisories</i>	69
<i>Figure 4.8: The simulation cycle of version 2.3</i>	71
<i>Figure 4.9: Execution time of different versions</i>	72
<i>Figure 5.1: Collaborative concurrent interactions</i>	74
<i>Figure 5.2: An example of collision</i>	77
<i>Figure 5.3: The ownership services used in our prototype system</i>	79
<i>Figure 5.4: The middleware with Interaction Resolvers</i>	81
<i>Figure 5.5: Algorithm of IRs in the prototype system</i>	83
<i>Figure 5.6: How consistency is ensured using IRs in our game</i>	84
<i>Figure 5.7: The use of enlarged regions with IRs</i>	85
<i>Figure 5.8 The minefield for picking up the same mine test</i>	87
<i>Figure 5.9: The test cycle of test one using OM</i>	88
<i>Figure 5.10: The test cycle of test one using IRs</i>	90
<i>Figure 5.11: Execution time of OM and IRs in picking up the same mine</i>	92
<i>Figure 5.12: Execution time of OM in picking up the same mine</i>	92
<i>Figure 5.13: Execution time of IRs in picking up the same mine</i>	93
<i>Figure 5.14: Execution time of OM and IRs for 2 agent federates</i>	93
<i>Figure 5.15: Execution time of OM and IRs for 3 agent federates</i>	94
<i>Figure 5.16: Execution time of OM and IRs for 4 agent federates</i>	94
<i>Figure 5.17: The minefield for stepping to the same grid cell test</i>	95
<i>Figure 5.18: The test cycle of test two using OM</i>	96

<i>Figure 5.19: The test cycle of test two using IRs</i>	<i>98</i>
<i>Figure 5.20: Execution time of OM and IRs in stepping to the same grid cell</i>	<i>100</i>
<i>Figure 5.21: Execution time of OM in stepping to the same grid cell.....</i>	<i>100</i>
<i>Figure 5.22: Execution time of IRs in stepping to the same grid cell.....</i>	<i>101</i>
<i>Figure 5.23: Execution time of OM and IRs for 2 agent federates</i>	<i>101</i>
<i>Figure 5.24: Execution time of OM and IRs for 3 agent federates</i>	<i>102</i>
<i>Figure 5.25: Execution time of OM and IRs for 4 agent federates</i>	<i>102</i>

LIST OF TABLES

<i>Table 2.1: Differences between analytic simulations and virtual environments.....</i>	<i>9</i>
<i>Table 3.1: Object class attribute publications and subscription in prototype.....</i>	<i>49</i>
<i>Table 3.2: Interaction class publications and subscription in prototype</i>	<i>50</i>
<i>Table 4.1: The DDM regions of version 2</i>	<i>63</i>
<i>Table 5.1: Distribution of agents in federates</i>	<i>91</i>

CHAPTER

1 INTRODUCTION

1.1 Background

1.1.1 Agents

Agents (adaptive or intelligent agents and multi-agent systems) are one of the most prominent and attractive technologies in computer science at the beginning of this new century. According to Wooldridge [1], an *agent* can be regarded as an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.

An agent can receive inputs from the environment through its sensor and act on the environment through its effector. It may also communicate with other agents via some form of communication language and typically has the ability to engage in collaborative problem solving. Multi-Agent Systems (MAS) refer to systems in which many intelligent agents interact with each other.

The autonomy, social ability, reactivity and pro-activeness of agents offer great flexibility in various situations, thus agents and multi-agent systems are being used increasingly in a wide range of application areas, including information retrieval, telecommunications, business process modeling, education, military simulations,

social simulations, games etc. Recently, there is a trend of using agents in distributed simulations. There are various research issues focused on this topic [2]. Most commonly, due to the limitations of current development tools and methodologies of agent systems, simulation is used to help agent system developers learn more about agents' interactive behaviors and investigate the implications of alternative architectures and coordination strategies. Also some researchers use agents to control simulations or provide advanced simulation services. The novelty of our project is to use agents in distributed simulations, representing some of the entities in Distributed Virtual Environments (DVEs).

1.1.2 Distributed Virtual Environments

Scientists and engineers have long relied upon modeling as a means to represent the salient characteristics of complex systems or phenomena. With the use of modeling, the concept of simulation has evolved. In simple terms, simulation is defined as the process in which the dynamic behavior of one system (the original) can be predicted or extrapolated by observing the behavior of another less complex system (the model).

Conceptually, computer simulation is a computer program representing and emulating the behavior of a physical system over time. In recent years, distributed simulation has become increasingly important as a strategic technology that links simulation components of various types at geographically different locations to create a common virtual world. With the development of networking technologies and computer graphics, Distributed Virtual Environments (DVEs) have become a major area of interest. A networked virtual environment is a distributed simulation of a virtual world in which multiple users interact with each other in real-time, even though these users (represented by *avatars*) may be physically located in different geographical places. An environment usually aims to provide users with a sense of realism by incorporating 3D graphics and sound to create an impressive experience, but it can also have different user interfaces, such as a text interface, or a graphical user interface (GUI). This technology is widely used in many areas, such as military battlefield simulation, E-education, business, etc.

As an example, in a military simulation, people situated in different countries in the world can participate in the same virtual battlefield, where they can act as if they were in a real war scenario: A German player of a tank can see its surrounding information, and fire at an aircraft within its range, no matter where the player of that aircraft is situated.

Singhal and Zyda describe the main features of networked virtual environments as follows [3]:

- *A shared sense of space*: All users are presented with the illusion of being in the same place.
- *A shared sense of presence*: Each participant is represented by a virtual representative, called an *avatar*. When a participant enters a virtual environment, he/she can see other participants' avatars and others can see the new participant's own avatar.
- *A shared sense of time*: Multiple users, located in different physical locations, interact with each other in real time. In other words, users should be able to see each other's behavior as it occurs.
- *A way to communicate*: Virtual environments also strive to enable some sort of communication among participants. This communication may occur by gesture, by typed text, or by voice.
- *A way to share*: Users should be able to interact not only with each other but also with the virtual environment itself.

1.2 Motivation

As mentioned above, the motivation for our effort is to use agents in distributed simulations, representing some of the avatars in DVEs. The characteristics of agents can be used in modern DVEs, where several avatars can act autonomously according to the latest information about the environment in which they participate, and thus act on the environment. When agents join the simulation, the simulation environment has to meet a variety of requirements. Simulation techniques are required that combine a flexible model design with efficient execution. Computational requirements of simulations of agent-based systems are often beyond the capabilities

of a single platform. Each agent typically requires considerable computational resources, and a number of agents may be needed to investigate the behavior of the system as a whole. These heavy workloads demand high performance computation environments that support loosely coupled, distributed, and concurrent simulation executions. This is definitely a challenge for current research of large-scale Distributed Virtual Environments.

Multi-agent simulation models should also be able to interact with other simulation models. This leads us to focus on combining agents and simulation, i.e. employing agents to execute in simulation models, and to implement flexible state-of-the-art simulation systems. The High Level Architecture (HLA) [4] is a standard for such a simulation system that can combine existing simulation models with new simulation models. It is deliberately designed to support the reusability and interoperability of component simulators.

In our project, we will utilize the benefits of HLA and JADE (Java Agent DEvelopment Framework) agent platform to investigate the simulation of multi-agent systems. JADE [5,6] is a software framework fully implemented in the Java language. It simplifies the implementation of multi-agent systems with both a middle-ware that complies with the Foundation for Intelligent Physical Agents (FIPA) specifications [7] and a set of tools that support the debugging and deployment phase. The agent platform can be distributed across machines and the configuration can be controlled via a remote GUI.

1.3 Objectives

The main objective of this project is to support the distribution of agents across multiple processors and avoid the bottlenecks that result from a single processor executing all agents. To achieve efficient performance and scalability of the distributed system, two main research issues are investigated, and they are:

- *Using interest management to construct sensors of agents:* Agents make deliberations and act based on information in their sensors. The acquisition of the information across distributed simulations can be realized using interest

management schemes, which are used to filter out unnecessary data and reduce the network traffic in the simulation.

- *Resolving concurrent interactions*: Management of the shared environment is the key problem of this system, as the shared environment needs to be structured in such a way as to allow agents to interact with the environment in a causally consistent way. There will usually be either mutually exclusive or collaborative interactions in agent based distributed simulations. To solve the effect of these interactions and still guarantee the consistency and causality of the system, some middleware components should be devised.

As described in *Motivation*, we will utilize the benefits of HLA and the JADE agent platform in this project.

1.4 Organization of the Thesis

This thesis is organized in six chapters.

Chapter 2 provides an overview of related work in Distributed Virtual Environments. These topics include different classes of simulation system, the structure of the environment, group multicasting, partitioning schemes, an introduction to the High Level Architecture (HLA), ownership management, time management and interest management. This chapter also includes a discussion on the relationship between simulations and agents.

In Chapter 3, the design and a simple implementation of our prototype, a minesweeping game, is presented. This implementation is the foundation of our efforts to investigate the distributed simulation of agent-based systems. Thus there is only one agent in the environment. There will be more agents joining the game in Chapters 4 and 5.

Chapter 4 is mainly concerned with how to use interest management to construct the sensors of agents. Due to the highly dynamic property of agents, an enlarged subscription region algorithm is proposed to avoid the causality problem caused by

overdue information. Three different implementations of interest management in our prototype are carefully described and compared. Experimental results are given to show the most efficient implementation.

Concurrent interactions are discussed in Chapter 5. We develop a middleware component called *Interaction Resolvers (IRs)* for all federates. Two different approaches to resolve concurrent interactions using ownership management services provided by the HLA and IRs are discussed in detail based on mutual exclusion problems in the game. Experimental results are given to show that our IRs are more efficient than ownership management services.

Finally, Chapter 6 concludes the thesis and summarizes benefits of this project for research areas in distributed simulation of multi-agent systems. Proposals for future research are also given in this chapter.

CHAPTER

2 RELATED WORK

A principal goal of most Distributed Virtual Environments is to achieve a “sufficiently realistic” representation [8] of a real system as perceived by its participants (*avatars*). Although every aspect of the environment does not have to be absolutely accurate in its minutest detail, research into large-scale Distributed Virtual Environments has been limited because of a number of practical factors; these factors involve research issues in network technology, software development and computer graphics.

As the number of avatars and the size of the network increases, there are two major issues in DVE: *consistency* and *communication cost*. Consistency refers to the requirement that state information of an avatar or an object be updated in all DVE clients within a certain tolerable threshold. Normally in order to achieve a consistent view, DVE clients have to exchange information among them. This incurs a significant *communication cost* to the underlying network, because the state information exchange is done periodically and usually the interval is small. The ideal DVE system is to provide a perfect consistent view, with very low communication cost.

This chapter begins by discussing two different classes of simulation application and gives an introduction to the High Level Architecture (HLA). Various techniques concerned with the DVE including the structure of the environment, group multicasting, partitioning schemes, ownership management, time management and interest management will also be discussed. Finally in this chapter, the relationship between simulations and agents will be introduced.

2.1 Analytic Simulations versus Virtual Environments

There are two popular classes of simulation applications: *analytic simulations* and *virtual environments* [8].

Analytic simulation is the classical approach to simulation; it has been used since the computer was invented. This kind of simulation usually attempts to capture detailed quantitative data concerning the system being simulated, and the ordering of events should be precisely reproduced. It has limited or no interaction with the outside world during simulation execution, and the simulations are executed “as-fast-as-possible” which could mean the simulator can advance faster than real-time or slower than real-time. For example, to investigate the air traffic of an airport in 24 hours, we can have an analytic simulation of that airport, and the execution time for this simulation can be only minutes instead of 24 hours. But all the detailed behaviors of this airport have to be emulated to produce exact statistical results and the arriving and departing events have to be ordered correctly.

Virtual environments have been introduced in Chapter 1. They differ from traditional analytic simulations in several ways. Instead of just observing or controlling the simulation as is the case in analytic simulations, human participants or physical devices act as entities in the simulation. Most virtual environments try to give users the look and feel of being embedded in the system being modeled. The simulation time in virtual environments needs to be advanced at approximately the same rate as the time perceived by the human participants. Moreover, there is no need to simulate the real system so exactly. It is usually acceptable for there to be differences between the simulated world and the actual world, as long as they are not perceptible to

human participants. For example, if two events occur almost at the same time so that the human cannot tell which occurred first, it may be acceptable for the simulated world to model these events in a reversed order that differs from the real system. Table 2.1 from [8] shows the main differences between the two kinds of simulations.

Table 2.1: Differences between analytic simulations and virtual environments

	Analytic Simulations	Virtual Environments
Execution pacing	As-fast-as-possible	Real-time
Typical objective	Quantitative analysis of complex systems	Create a realistic and/or entertaining representation of an environment
Human interaction	If included, human is an external observer to the model	Humans integral to controlling the behavior of entities within the model
Before-and-after relationships	Attempt to precisely reproduce before-and-after relationships	Need only reproduce before-and-after relationships to the extent that humans or physical components embedded in the environment can perceive them

2.2 Structure of the Environment

2.2.1 System architecture

There are three popular methods to distribute data in a virtual environment.

2.2.1.1 Centralized server architecture

In a *centralized* architecture, one computer collects all of the data from the different machines, stores the changes in some collection of data structures (the centralized

database), and then sends the results back to each participating machine. Each machine then renders the scene and handles user inputs. This is often referred to as a logical star-configuration, as Figure 2.1 shows.

Figure 2.1: A centralized server architecture

It is clear that the centralized architecture has the following advantages:

- Global consistency is guaranteed as all avatars receive the same state information from the server.
- A natural synchronization among players is brought by the centralized server.
- Management of the clients is easier with the use of a centralized server.

However, it is also very obvious that there are limits to this model:

- *Lack of robustness.* If the server crashes, all the clients are lost and the simulation is over.
- *Lack of scalability.*
 - The server performs all the computations.
 - The server concentrates all the network traffic of the simulation.

These two points limit the acceptable complexity of the simulation and the number of participants.

- *Two step communication for interaction:* client-server for actions and server-client for resulting state information.

From the last point, it is noticed that there is no real-time synchronization between clients. Clients have to wait for global state from the server to display it. Note that every client receives the same state information, but at a time depending on the transmission delays.

The centralized server architecture is useful and efficient for small simulation systems, but as more processes enter the simulation, the computational requirement rises dramatically and the centralized sever will be a bottleneck for the whole system.

2.2.1.2 Distributed serverless architecture

An alternative, more scalable approach is to incorporate a *distributed* architecture. Each player maintains its own complete, local copy of the database as well as performing the rendering, computation and animation of objects. When a player makes changes to its own database, it sends the update data out, so that other programs can update their individual databases. This changes the scalability problem from being a CPU bottleneck (in the star configuration) to one where there are many connections and messages. Figure 2.2 shows this architecture.

Figure 2.2: A distributed serverless architecture

To reduce the number of connections and the number of messages being sent, three communication techniques may be employed: network broadcast, multicast and dead reckoning:

- *Network broadcast*: Broadcasting is a protocol that allows a player to send one message out, and all of the other players to read that single message. This significantly reduces the number of messages being sent during database updates. Using broadcast also simplifies the programming, since a player joining a simulation in progress does not have to establish many point-to-point connections with the other processes. Broadcast is used in SIMNET [9] and many Distributed Interactive Simulation protocol (DIS) [10] systems. It is adequate for small systems, but it does not scale because communication increases dramatically with more simulators.
- *Multicast*: Other than broadcast, the network provides a mechanism to only deliver messages to members of the multicast group. This is often accomplished by constructing a spanning tree that controls the generation and distribution of copies of the message among the members of the group. Alternatively, a flooding/pruning mechanism may be used where intermediate network nodes retransmit copies of incoming messages on outgoing links according to certain rules that guarantee distribution to all destination nodes but economize on bandwidth usage. In [11], an architecture is described that logically partitions the virtual environment by associating spatial, temporal, and functional classes with network multicast groups. Environment partitioning will be introduced in the next section.
- *Dead reckoning*: If there are a large number of players in the DVE, there will be thousands of messages to be exchanged using network broadcast. The *dead reckoning* algorithm is used to reduce such inter-processor communication. It is essentially a technique for synchronization of the state information of spatial objects in different computer systems. The basic idea is that, rather than send frequent state updates, each simulator estimates the location of remote entities from its last reported position, direction, velocity, and the like [8]. It is the heart of some simulation systems, such as DIS, and is used in SIMNET and NPSNET [12]. Dead reckoning was proposed at first to

reduce communication load brought by broadcast in DIS, and now it can be used with both broadcast and multicast in various simulation systems. For more information about dead reckoning, see [8,13].

2.2.1.3 Hybrid architecture

In this architecture, the entire virtual environment is divided into several partitions; there is a server responsible to handle a subset of the virtual environment. Several servers are therefore used to reduce the heavy workload of the single server in the centralized architecture. These servers pass the data to other servers through peer-to-peer multicast messaging, while the players (clients) connect to their servers using client-server networks [14]. With proper division of the whole DVE into partitions, the workload of the whole DVE can be nicely shared and the inter-server communication cost incurred due to partitioning can be minimized as well. Figure 2.3 shows this architecture.

Figure 2.3: The hybrid architecture

2.2.2 Partitioning the virtual environment

In the distributed and the hybrid architecture, by dividing the DVE into partitions, the workload of the whole system is distributed to more than one machine. However, the division introduces communication cost into the system, thus we have to balance between shared workload and the overhead due to the inter-server communication.

The partitioning problem is to divide the whole DVE into partitions such that the workload of the whole DVE is nicely shared among the DVE servers and the inter-server communication cost is minimized. The partitioning schemes can follow one of the restrictions below:

- Cell-based partition: Each DVE server is responsible for a partition that is a region formed by a set of cells. Suppose a cell size is equal to $L \times L$, then the size of each partition must be a multiple of L , as shown in Figure 2.4.

Figure 2.4: Cell-based partition

- Non-cell-based partition: the size of the partition is not restricted by the cell size, and it can be any shape and size as Figure 2.5 shows.

Figure 2.5: Non-cell-based partition

Macedonia et al [11] suggest three possibilities for partitioning the scene, and associating these classes with multicast groups:

- *Spatial Partitioning*: This is based on partitioning the scene into areas, which can be processed in parallel and independently. Therefore, the participants existing within the same part of the virtual world can interact with each other, i.e. message exchange occurs with only a subset of remote participants.
- *Temporal Partitioning*: Some entities might require real-time update of all entities within a certain time period, while other entities might require updates within another time period. For example, there may exist some entities that are responsible for the system management, and they need updates every few minutes in order to have an overall view of the virtual environment. In essence, objects that require different rates of updates can be systematically grouped together. Groups requiring a higher update rate can then have a larger share of the total network bandwidth.
- *Functional Partitioning*: Entities may belong to a functional class in which an entity may communicate with a subset of entities. For example, a radio communication message can only be sent to the entities having radio receivers. Other types of functional class could be system management or services such as time.

There are various possible schemes for partitioning the virtual environment. A choice for distribution may depend on the application, client architecture, network topology and other design decisions. References [15,16,17,18,19] provide more information about the partitioning.

2.3 High Level Architecture

2.3.1 An Overview of the HLA

Many large and complex simulations involve a combination of simulations of several different types of system, to make up the total environment to be simulated. Often simulations of some of these components already exist, and although they may be developed for a different purpose, they could be used in the new simulation. Unfortunately extensive modifications are necessary to make these simulations adapt for use with other components. In some cases it may prove easier to implement a completely new simulation of a system component than it is to modify an existing one. In other words traditional simulation models often lack two desirable properties: *reusability* and *interoperability*.

- *Reusability*: Component simulation models can be reused in different simulation scenarios and applications.
- *Interoperability*: The reusable component simulations can be combined with other components without the need for re-coding. It implies an ability to combine component simulations on distributed computing platforms of different types, often with real-time operation.

The High Level Architecture (HLA) [4,20,21] provides a standard that will reduce the cost and development time of simulation systems and increase their capabilities by facilitating the reusability and interoperability of component simulators. By adhering to a common standard, simulators need not be limited to the application for which they were specifically designed and will be available for use in other simulation systems.

The HLA is a current U.S. Department of Defense (DoD) and an industry (IEEE-1516) standard architecture [22,23,24] for modeling and simulation. It was developed

by the Defense Modeling and Simulation Office (DMSO) of DoD to meet the needs of defense-related projects. It is increasingly being used in various simulation application areas, including education, training, analysis, engineering, entertainment and games, representing entities at many levels of resolution.

In the HLA a distributed simulation is called a *federation*, and each individual simulator is referred as a federate, one point of attachment to the RTI. A federate can be a computer simulation, it can also be an instrumented physical device or a passive data viewer. An HLA federation is made up of [21]:

- *One or more federates.*
- *A Federation Object Model (FOM):* The FOM defines the types of and the relationship among the data exchanged between the federates in a particular federation.
- *The Runtime Infrastructure (RTI):* The RTI software implements the interface specification and provides services in a manner that is comparable to the way a distributed operating system provides services to applications.

Figure 2.6 illustrates the software components in an HLA federation.

Figure 2.6: Software components of an HLA federation

As stated above, each federate has a single point of contact with RTI, the single point of contact is the definition of a federate from the RTI's perspective. No matter how many computers one federate is running on, it maintains only one connection to the RTI, and it can only communicate with other federates through the RTI.

Figure 2.7 shows the names of the interfaces between each federate and the RTI. Each federate provides the RTI with an interface called *FederateAmbassador*, the RTI can invoke operations on that *FederateAmbassador* when it calls the federate. The RTI also offers each federate an interface named *RTIambassador*, and the federate can invoke operations on *RTIambassador* to request services of the RTI. Thus some RTI services are defined as part of the *RTIambassador* interface, and some are defined as part of the *FederateAmbassador* interface.

Figure 2.7: Interfaces between RTI and federates

As a standard, HLA is defined by three components (1) Federation and Federate Rules, (2) the HLA Interface Specification, and (3) the Object Model Template (OMT).

(1) Rules: The Rules [22] describe the responsibilities of federates and their relationships with the RTI. There are ten rules. Five relate to the federation and five to the federate.

Federation Rules:

1. Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT).
2. In a federation, all representation of objects in the FOM shall be in the federates, not in the run-time infrastructure (RTI).
3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.

4. During a federation execution, federates shall interact with the run-time infrastructure (RTI) in accordance with the HLA interface specification.
5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

Federate Rules:

6. Federates shall have an HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).
7. Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.
8. Federates shall be able to transfer and/or accept ownership of an attribute dynamically during a federation execution, as specified in their SOM.
9. Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM.
10. Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

(2) The HLA interface specification: The interface specification [23] identifies how federates will interact with the federation and, ultimately, with one another. The Interface Specification describes six service classes for supporting HLA federations:

- *Federation management:* Manages creation, dynamical control, modification and deletion of a federation execution.
- *Declaration management:* Includes publication, subscription, and supporting control functions, and provides information on data relevance at the *class* attribute level.
- *Object management:* Deals with the registration, modification, and deletion of object instances and the sending and receiving of interactions.
- *Ownership management:* Manages transfer of instance attribute ownership between federates.
- *Time management:* Controls the advancement of simulated (logical) time.

- *Data distribution management*: Adds the capability to further refine the data requirements at the *instance* attribute level, based on the abstract regions of routing spaces.

(3) The Object Model Template: All objects and interactions managed by a federate, and visible outside the federate, are described according to the standard OMT [24]. The OMT provides a common method for representing HLA Object Model information. The HLA separates data and architecture. It prescribes that OMT objects and interactions defined according to the OMT can be constructed and exchanged with no adjustments to HLA-derived software. For more information about HLA, see [4, 20, 21, 25].

Figure 2.8 from [25] depicts the HLA Federation Development and Execution Process (FEDEP) Model, which is intended to identify and describe the sequence of activities necessary to construct HLA federations [25]. It illustrates the major activities that should take place during the life cycle of a federation. This model starts with the definition of federation objectives through the federation development and concludes with the results of a running a federation execution.

Figure 2.8: The HLA federation development and execution process model

2.3.2 Ownership Management in the HLA

Ownership management is used by federates and the RTI to transfer ownership of instance attributes among federates. It is possible for two or more federates to share update responsibility for a single object instance, however, only one federate can have update responsibility for an individual attribute of an object at any given time. Also only one federate has the privilege to delete an object instance at any given time [4].

There are two different methods for transferring attribute ownership among federates, namely, *push* and *pull*.

- *Push Model*: A federate that owns the ownership of one or more attributes of an object instance can initiate the ownership transfer by telling the RTI its willingness to give up the ownership.
- *Pull Model*: A federate that wants to get ownership of one or more attributes of an object instance can send a request to the RTI to initiate the process.

The *push* can be done in two different ways, unconditional or negotiated [4]. In the “unconditional push”, the owner federate informs the RTI that it will give up the ownership whether or not any other federate is interested in acquiring the ownership. The federate can call *unconditionalAttributeOwnershipDivestiture()* to release the ownership. It is highly possible that those attributes will be left unowned (orphaned-attributes). On the contrary, in the “negotiated push”, the RTI will make sure there are interested federates in the federation before it releases the ownership of a specific set of attributes. So there would be no orphaned-attributes in the federation. Figure 2.9 illustrates the process of a “negotiated push”. The federate wishing to release responsibilities calls *negotiatedAttributeOwnershipDivestiture()*. Federates that are capable of publishing any or all of the attributes being given away are notified via the *requestAttributeOwnershipAssumption()* callback (callbacks are highlighted with a * in all figures). A federate wishing to acquire one or more of the offered attributes makes use of the pull methods: *attributeOwnershipAcquisition()* or *attributeOwnershipAcquisitionIfAvailable()*. If federates are found that are willing to assume the responsibilities being given away, the federate that initiated the push

receives the callback *attributeOwnershipDivestitureNotification()*, which informs the federate that it is no longer responsible for the listed attributes. The federates gaining responsibility for the attributes are informed of their new responsibility with the callback *attributeOwnershipAcquisitionNotification()*.

Figure 2.9 : Diagram of negotiated push

For the *Pull*, there can be both “obtrusive pull” and “unobtrusive pull”. The former method attempts to secure ownership of an attribute whether it is owned by another federate or not. When a federate sends a request, the owner federate will receive a *requestAttributeOwnershipRelease()* callback. The owner federate can respond with *attributeOwnershipReleaseResponse()* to tell the RTI that it agrees to release the ownership. The requesting federate will get an *attributeOwnershipAcquisitionNotification()* callback and become the new owner. Figure 2.10 shows the process an “obtrusive pull”.

Figure 2.10: Diagram of obtrusive pull

In the “unobtrusive pull”, the requesting federate will use `attributeOwnershipAcquisitionIfAvailable()` to secure attributes that are either not owned or being offered for divestiture by another federate. When it is called, any unavailable attributes will be reported via the RTI using the `attributeOwnershipUnavailable()` callback. If attributes are available, the requesting federate will receive an `attributeOwnershipAcquisitionNotification()` callback. “Unobtrusive pull” ensures all federates involved in the ownership transfer be notified of the result of their requests. While in the “obtrusive pull”, the requesting federate will have to keep on waiting for the ownership until the attributes are available. Figure 2.11 provides a diagram that illustrates the “unobtrusive pull”.

Figure 2.11: Diagram of unobtrusive pull

Some problems and limitations of the HLA ownership management services are discussed in detail in [26,27]. The authors summarize their experience of ownership management services in the context of their air traffic control simulation. Four main problems are presented: oscillation effects, pending requests, lack of a proper three way handshaking procedure and lack of a way to select a successor of the ownership. For the last problem, they propose a general directed ownership transfer mechanism.

2.3.3 Time Management in the HLA

Since simulation represents or emulates the behavior of another system over time, it is easy to be confused with the concept of “time” in simulations. Basically, there are three different types of time in distributed simulations [8], they are:

- *Physical Time*: The time in the physical system being modeled.
- *Simulation/Logical Time*: The simulator’s representation of the time.
- *Wallclock Time*: The real-world time during the execution of the simulation program.

In the HLA, time in the system being modeled is represented in the federation as points along a federation time axis. Each federate, upon joining an execution, is assigned a *logical time*. A federate’s logical time initially is set to the initial time on the federation time axis (time zero). Time within a federation only advances; thus a federate may request to advance only to a time that is greater than or equal to its current logical time. For a federate to advance its logical time, it requests an advance explicitly. The advance will not occur until the RTI issues a grant. Time management is exactly concerned with the mechanisms for controlling the advancement of each federate along the federation time axis. In general, time advances are coordinated with object management services so that information is delivered to federates in a causally ordered fashion.

The manner in which current HLA services are coordinated with time is through the concept of *messages*. Messages sent by one federate typically result in one or more other federates receiving a corresponding message. Each message, sent or received, shall be either a time-stamped order (TSO) message or a receive order (RO)

message. RO messages are simply put in a FIFO queue of a federate when they arrive, and are ready to be delivered to the federate immediately. The ordering of these messages is arbitrary. On the other hand, in the time stamp ordering, each event in the simulation is assigned a time stamp, and messages are delivered to each federate in order of non-decreasing time stamps. The time stamp ensures the before-and-after relationships are properly modeled by federates, and assuming the events with the same time stamp are handled in the same order by all federates, it also ensures that all participating federates see the same ordering of events. Time stamp ordering can be regarded as a sequential consistency model, as the time stamps ensure the sequence of all events. In the HLA, only the following events can be time stamped:

- *updateAttributeValues()* and its callback *reflectAttributeValues()*.
- *sendInteraction()* and its callback *receiveInteraction()*.
- *deleteObjectInstance()* and its callback *removeObjectInstance()*.

There are commonly two types of discrete simulations: *time-stepped* and *event-driven* simulations. The main difference lies in the way they request to advance their logical time. For time-stepped simulations, simulation time is divided into time steps of the same size, and the simulation advances from one time step to the next. A simulation program will calculate a new state for the simulation in each time step and advance to its next step, even though it is not necessary to do the calculation every time step. Actions happening at the same time step are usually treated as concurrent as they have the same logical time. Thus special attention should be given to those actions that have causal relationships, and the size of the time step is also important as it determines the precision of the simulation. In the HLA, Time Advance Request (TAR) is normally used in this kind of simulation. The federate invokes this service to request its logical time to be advanced to the next time step. All TSO messages with time stamp less than or equal to the time requested will be delivered to this federate.

On the other hand, rather than advancing from one time step to the next, simulation time in event-driven simulations can advance from the time stamp of one event to

that of the next event. Here *event* is an abstraction used in the simulation to model some actions in the physical system. Each event has a time stamp which stands for the time the event will occur. Events usually change state variables in the simulation. So instead of computing a new value for state variables each time step, event-driven simulations only need to update the variables at the time of the next event on its queue. Next Event Request (NER) is used in the HLA for those simulations. When a NER request is used, all TSO messages that have a time stamp equal to the minimum next event time of any message will be delivered.

To avoid deadlock [8], all the time-advance mechanisms in the RTI use a *lookahead* value. *Lookahead* is a duration of logical time that each federate specifies when it becomes time-regulating. It can be altered dynamically, and it is also possible to set its value to zero. *Lookahead* establishes the earliest possible TSO event the federate can generate relative to the current time, and it places a restriction on the time-regulating federate: the federate must look ahead from its current logical time and not send events with time stamp less than its current time plus its lookahead.

2.4 Approaches to Interest Management

2.4.1 Introduction

The problem of reducing broadcast communication has been addressed mainly in the context of real-time large scale simulations where it is termed Interest Management (or equally, *data distribution management*, *relevance filtering* and *data subscription*) [28]. Interest Management techniques utilize filtering mechanisms based on *interest expressions* (IEs) to provide the components in the simulation with only that subset of information which is relevant to them (e.g., based on their location or other application-specific attributes). The data of interest to a component is referred to as its *Domain of Interest* (DOI) or *Area of Interest* (AOI). Special modules in the simulation, referred to as *Interest Managers*, are responsible for filtering generated data and forwarding it to the interested components based on their IEs [28]. The region of the multi-dimensional parameter space in which an Interest Manager is responsible for managing data transmission is referred to as its *Domain of Responsibility* (DOR).

Various Interest Management schemes have been devised, utilizing different communication models and filtering schemes [28]. In most existing systems, Interest Management is realized via the use of IP multicast addressing, whereby data is sent to a selected subnet of all potential receivers. A multicast group is defined for each message type, grid cell (spatial location) or region in a multidimensional parameter space in the simulation. Typically, the definition of the multicast groups of receivers is static, based on a priori knowledge of communication patterns between the components in the simulation [11,29,30,31,32]. For example, the High Level Architecture (HLA) utilizes the *routing space* construct, a multi-dimensional coordinate system whereby simulation federates express their interest in receiving data (subscription regions) or declare their responsibility for publishing data (update regions) [4]. In some existing HLA implementations, the routing space is subdivided into a predefined array of fixed size cells and each grid cell is assigned a multicast group which remains fixed throughout the simulation; a federate joins those multicast groups whose associated grid cells overlap the federate subscription region.

Static, grid-based Interest Management schemes have the disadvantage that they do not adapt to the dynamic changes in the communication patterns between the components during the simulation and are therefore incapable of balancing the communication and computational load, with the result that performance is often poor. Furthermore, in order to filter out all irrelevant data, grid-based filtering requires a reduced cell size, which in turn implies an increase in the number of multicast groups, a limited resource with high management overhead. Some early systems, such as JPSPD [11] and STOW-E [33] did exhibit some degree of dynamism in their filtering schemes. More recently, there have been a few attempts to define alternative dynamic schemes for Interest Management concentrating mainly on the dynamic configuration of multicast groups within the context of HLA. For example, Berrached et al. [34] examine hierarchical grid implementations and a hybrid grid/clustering scheme of update regions to dynamically reconfigure multicast groups. A finer grid is used at the local network level and the coarse grid at the wide area network level. Calvin and Van Hook [35] propose filtering agents that can be used at the gateways to implement filtering at the wide area network level. Morse et

al. [36] report on preliminary investigations on a dynamic algorithm for dynamic multicast grouping for HLA. Saville et al. [37] describe GRIDS, a generic runtime infrastructure which utilizes dynamic instantiation of Java classes in order to achieve Interest Management. The Joint MEASURE system [38,39,40] is implemented on top of HLA and utilizes event distribution and predictive encounter controllers to efficiently manage interactions among entities. There are also cluster based filtering mechanisms. Cluster based filtering is based on grouping objects that have similar subscription regions and/or update regions [41]. Ideally, objects that are subscribing to the same update region are grouped in the same cluster and assigned a multicast group. Updates would then be routed only to the subscribers of the update region through the multicast group.

2.4.2 Interest management mechanisms in the HLA

The Data Distribution Management (DDM) services specified for the High Level Architecture (HLA) [4] are the latest in a succession of data reduction mechanisms for large scale distributed simulations.

The HLA supports two types of filtering:

- *Class-based filtering*: It uses Declaration Management (DM) services. DM services allow a federate to update and receive updates to object attributes based solely on object class.
- *Value-based filtering*: It uses Data Distribution Management (DDM) services. DDM services extend DM services using routing spaces and regions.

The fundamental concept used in the HLA to support value-based DDM is the *routing space*. A routing space is a normalised multidimensional coordinate system in which federates indicate interest in receiving or providing updates via *subscription* and *update regions*. Federates express their interest in receiving updates from other federates through *subscription regions*; they are called *subscribers* to a specific attribute. On the other hand, federates that are providing updates define their *update regions* over the routing space, and are called *publishers* of a specific attribute.

By calculating the intersection of update and subscription regions, the Run-Time Infrastructure (RTI) provides data transfer from the publisher to all subscribers for all updates to that attribute. Figure 2.12 shows an example of this mechanism.

Figure 2.12: Two-dimensional routing space example

2.4.2.1 Region based approach

The region-based approach is based on calculating the intersection of *regions*. In a federation where there are P number of update regions, and S number of subscription regions, at the start of the federation execution, a basic algorithm will perform $P*S$ matchings to see which regions overlap. After that, each publisher will keep a list of subscribers that have their regions overlapping with its region. When attribute updates are to be sent, the publisher uses its list of subscribers and multicasts to all of them, thereby reducing traffic flow. During the federation execution, at any time when one or more of the publishers change the region, the RTI has to perform $X*S$ matchings, where X is the number of publishers modifying their regions, to re-determine which update and subscription regions overlap. Figure 2.13 below illustrates an example of the basic region-based approach, in a simplified two-dimension routing space.

Figure 2.13: Two-dimensional routing space using the region-based approach

In this figure, the RTI sends data of Publisher P only to overlapping subscription regions S1, S2 and S3. But region S4, which does not overlap with P, will not receive any data.

Yu et al [42] present an improved, sort-based algorithm, which reduces the number of matchings the RTI has to perform. In their algorithm, regions are sorted before computing the intersections, which reduces useless checks and allows optimization at the implementation level.

2.4.2.2 Grid based approach

The DMSO RTI version 1.3 statically breaks up a routing space into grid cells, and assigns a channel to each cell [4]. The update and subscription regions are mapped into the grid cells. Entities simply subscribe to the channel correlating to the cells of the routing space in which they are interested. An update is sent out on all channels corresponding to cells that overlap the update region. As entities change interests, they change channels correspondingly.

Figure 2.14 below shows an example of how the grid-based approach works, in a simplified two-dimensional routing space.

Figure 2.14: Two-dimensional routing space using the grid-based approach

In this figure, if a federate updates its attributes with region P, all subscription regions S1, S2, S3, and S4 will receive the update no matter whether their regions intersect or not.

2.4.3 Comparison of different interest management schemes

There is a major difference between the region-based approach and the grid-based approach in the HLA. In the region-based approach, the RTI will update attributes for all subscription regions that overlap the update region of the attributes. However, in the grid-based approach, attribute updates will be sent out to cells that overlap the update region, and it is possible that there will be no intersection of the update and subscription regions. This situation is illustrated in Figure 2.15. The figure depicts a two-dimensional routing space subdivided into a 4 by 4 grid and two regions, S and U, which represent the subscription and update regions of federates F1 and F2, respectively. Note that S and U do not intersect and therefore data update from F2 should not be delivered to F1. However, because both regions overlap cell 9, any data update from F2 will be delivered to any subscriber to the multicast group associated with cell 9, including F1. Irrelevant data delivery can be decreased by

reducing the size of the grid cells. However, this would increase the number of multicast groups required and associated overhead.

Figure 2.15: Irrelevant data is delivered from F2 to F1 even though S and U do not intersect.

Also, in a region-based approach, when a region changes, the RTI must recheck the mappings of other regions to this region to redetermine which of them overlap. It is very time consuming, but given a small number of subscribers, the region-based approach will be efficient. By contrast, the fixed grid approach is relatively simple to implement and incurs little overhead. Its main feature is that no interaction between federates is needed to determine the required connectivity between senders and receivers. This is due to the fact that the grid is predefined and the regions occupied by the grid cells are regular (i.e. d-dimensional rectangles) and remain unchanged at run time. Matching of subscription and update regions to grid cells is therefore done by each federate locally. This improves the time efficiency for the RTI. So when there are large numbers of publishers and subscribers, the grid-based approach will be useful.

Both approaches have their own strengths and weaknesses. They have the same problem that some attribute updates may not lie in the intersecting areas, and this will cause the transfer of irrelevant data. For a region-based approach, as long as the intersection of an update region and a subscription region exists, the subscriber can

get all updates within that update region, and much irrelevant data are therefore transferred. In the normal fixed grid-based approach, irrelevant data can be delivered if a subscription and update region overlap the same grid cell, even if the attribute update does not lie in that cell. Ayani et al [43] have developed a grid-based DDM model that differs from the traditional one. In this model, the application user specifies the subscription and publication areas as regions, but the DDM maps these regions into cells. They associate two lists with each cell: a list of the cell's publishers and a list of the cell's subscribers. Based on these two lists, they identify the subscribers and publishers of each cell. Different from the region-based and traditional grid-based approaches, they assign only a portion of a publisher's region to a subscriber based on the intersecting cells, which reduces the transfer of irrelevant data. For example, in Figure 2.14, if a federate updates data in cell 33, it will be sent to S2 only, not to S1, S3 and S4.

There is another problem of determining the cell size. While too small a cell size will cause the region to be spilt into many cells, and increase the number of cells that have to be accessed when an area of the region is being modified, too large a cell size may cause entire regions to lie in one single cell, and the RTI may send a large amount of irrelevant data. The size of the regions created is also important. If the region is too small, then it occupies only one cell, and many small regions will cause the performance to depreciate to that of the region based approach. If it is too large, then the number of cells it occupies may increase, and thus the number of cells accessed for the updating of a region will increase. But this problem lies with the federates, as it is up to them to declare the size of the region they are publishing or subscribing. Thus for the grid-based approach to work, it has to take into account the size of each cell, and the size of the regions too.

The main drawback of static grid-based filtering is that no information about the dynamic state of the publishing or subscribing federates is used to optimize multicast group usage. Multicast groups are pre-allocated evenly to cells across the routing space and remain unchanged through out the simulation regardless of the needs of the federates. This results in inefficient utilization of multicast groups: low utilization

of multicast groups in sparsely “populated” regions of the routing space and insufficient multicast groups in the more heavily populated regions. For a more comprehensive analysis of fixed-grid based filtering see [44].

Tan et al [45] also outline a hybrid filtering approach in which they combine the advantages of region-based filtering and grid-based filtering. Experimental results show that their method produces less irrelevant data than that of the grid-based approach and reduces matching cost compared to the region-based approach.

2.5 Agents and Simulations

There are mainly three research areas in combining agent technology with parallel/distributed simulations.

1. Using simulation to understand agents’ behaviors.
2. Using agents for specific simulation services.
3. Using agents to represent entities in a simulation.

Most commonly, due to the limitations of current development tools and methodologies of agent systems, simulation is used to help agent system developers learn more about agents’ interactive behaviors and investigate the implications of alternative architectures and coordination strategies. For example, the JAMES (Java-based Agent Modeling Environment for Simulation) system developed by Uhrmacher et al [46] uses the parallel DEVS framework to model mobile, deliberative agents. They propose an approach which splits simulation and external agent deliberation into different threads, thus simulation and deliberation can proceed concurrently by utilizing simulation events as synchronization points.

Logan et al propose [47] a new approach to dynamic interest management, which targets the simulation of agent-based systems. The approach is not confined to grids and rectangular regions of multidimensional parameter space and does not rely on the support provided by the TCP/IP protocols. Rather, it is based on the notion of *spheres of influence*, and it is used to dynamically decompose and distribute the shared state so that bottlenecks and broadcast communication are minimized. In a later paper, Lees et al [48] outline an approach to the distributed simulation of agent-

based systems using the SIM_AGENT toolkit and the HLA. Using a simple Tileworld scenario as an example, they show how the HLA can be used to flexibly distribute a SIM_AGENT simulation with different agents being simulated on different machines. They also outline the changes necessary to the SIM_AGENT toolkit to allow integration with the HLA, and briefly describe the simulation cycle of the combined system called HLA_AGENT. The integration is transparent in the sense that the existing SIM_AGENT code runs unmodified and the agents are unaware that other parts of the simulation are running remotely. However, their approach has the drawback that they use a number of TARs for each simulation cycle, which will influence the performance of a distributed simulation.

In research area 2, some researchers use agents to control simulations or provide advanced simulation services. For example, Tan and Xu [49] use intelligent agents to perform data filtering in distributed simulations. They propose an agent-based DDM filtering mechanism to use agents to perform accurate data filtering in DDM. The agent-based DDM can filter out all the irrelevant information and route only the information that subscribers want.

Valentino et al [50] have developed a series of intelligent software agents, collectively referred to as *AgentsTools*, for the express purpose of aiding the establishment, execution, and monitoring of distributed simulations, including HLA/RTI compliant simulations. They have formulated agents that will assist the simulation user in setting-up, managing, and analyzing distributed simulations.

In research area 3, Andersson et al [51] add a KQML-layer to every federate which hosts an agent, in order for the agents to successfully communicate with each other. Other functionality is needed to keep track of remote agents' capabilities in their architecture. They propose a middleware to contain the features described above. They conclude that if the HLA/RTI is extended with an agent-specific services middleware, it would provide a suitable environment for intelligent agents. In their model, non-agent federates directly access the RTI; while agent federates do so indirectly via the agent middleware. However, no further references to an existing

simulation for this proposal have been found that provide an evaluation of the performance, especially when integrating autonomous agents into an HLA simulation.

Minson et al [52] intend to realize true interoperability of sequential agent simulation platforms which would allow agents using different toolkits to transparently interact in common abstract domains. They design and implement a system that is capable of harnessing the computational power of a distributed simulation infrastructure with the design efficiency of an agent-toolkit. The system permits integration, through an HLA federation, of multiple instances of the Java-based lightweight-agent simulation toolkit RePast. For mutual exclusion problems, they use ownership management to implement a set of variable types which provide various semantic behaviors required by agent models, and they manage to form a strict hierarchy in their schedules to prevent possible errors in granting ownership.

The novelty of our project is to use agents in distributed simulations, representing some of the entities in Distributed Virtual Environments (DVEs). Moreover, we will outline a flexible middleware architecture to integrate JADE agents in HLA-based distributed simulation, with particular attention to the efficient use of the HLA services like object management services, data distribution management and ownership management services.

CHAPTER

3 PROTOTYPE SYSTEM

3.1 Purpose of the Prototype System

In order to investigate and implement new ideas into the research, we examined various simulation systems used by other research groups. The systems varied from battlefield systems, air traffic control systems [53], stone-picking robot [54] to maze games like the TileWorld [55], e-commerce models [56] and simulations of natural ecological phenomena [47].

Based on the considerations of multi-agent systems and distributed simulations, a prototype system named minesweeping game was eventually proposed. This prototype is intended to be a testbed to provide an exploration of different issues to support the distributed simulation of multi-agent systems and their environments. These issues are mainly concerned with how to integrate agents into distributed simulation, how to use interest management to construct the sensors of agents and how to solve concurrent interactions in such highly dynamic and complicated agent based distributed simulations.

The game is implemented using the JADE agent toolkit version 2.5 [5], DMSO RTI1.3NG-V6 and JAVA jdk1.3.1_02.

3.2 Description of the Minesweeping Game

The minesweeping game contains a certain number of soldiers represented by autonomous agents and an environment shared by the soldiers. The soldiers aim to make the environment area safe by clearing all the mines in it before they explode.

The environment has an $n*n$ grid, where n can be set by the user. There are a number of randomly distributed obstacles (trees and rivers) and mines in the environment. Alternatively, using a GUI, those obstacles and mines can also be set in the environment by the user.

The obstacles are static as they exist through out the simulation and cannot be moved by soldiers. In contrast, the mines are dynamic in that they can be picked up and cleared from the environment by soldiers. If a mine has not been cleared before its life expires, it will explode and the soldiers will have failed in their mission. The game is over when all the mines in the environment have been cleared by the soldiers. Figure 3.1 shows the environment with and without a soldier agent in it.

any object within its sensor region: if there are obstacles, it will step in another direction to avoid the obstacles, if there are mines, it will select a mine to pick up, add the mine to its package and then walk purposely towards the border of the environment to release the mine. After this, it will roam in the area again, trying to find another mine until all the mines have been cleared from the environment. Currently, it is presumed that when a soldier has a mine in hand, it cannot pick up another mine until it releases the mine in its hands at the border. So when a soldier detects a mine while its hands are full, it can only inform other soldiers nearby to come and pick it up.

Every soldier agent has a limited knowledge of the environment; they get the information via their own sensors, do deliberations and act upon the environment using their effectors. This process is illustrated in Figure 3.2.

Figure 3.2: Soldier agent architecture of the prototype system

According to this architecture, the simulation cycle of a soldier agent in this prototype system is divided into three logical phases: *Sensing*, *Deliberation* and *Action*. In the first phase of sensing, the agent gets data from the environment, then it will decide its corresponding actions according to the rules in the deliberation phase, and finally in the third phase, the agent will perform actions. The simulation cycle of a soldier agent in this prototype system is shown in Figure 3.3.

Figure 3.3: Logical structure of a simulation cycle of an agent

In order to simplify the system, two federates are initially used in this prototype: the environment federate and the soldier agent federate, representing the minesweeping environment and the soldier agent respectively. Furthermore, there is only one soldier agent residing in the JADE container. A more complex system with more agents is investigated and described in chapter 5.

One of the main problems of this prototype is how to construct the sensor of an agent. What an agent can get from the environment wholly depends on the configuration of its sensors. In general, sensors of agents can have different regions, and allow the agent to sense objects and other agents within a certain area centered on its current position. In this prototype, one size of sensor is used by an agent, and it can detect information about the nine cells centered on the agent's current cell in the environment. Figure 3.4 depicts the sensor region of the agent. The agent can choose to move into one of the eight cells around it if there is no obstacle around.

Figure 3.4: Sensor region

In this prototype, the environment for agents is divided into evenly distributed grid cells. These cells are used to identify the 2D position for the agents and obstacles. Also the grid can be used for data distribution services provided by the RTI, which is

aimed to reduce the communication load. The DMSO RTI version 1.3 statically breaks up a routing space into grid cells, and assigns a channel to each cell [4]. So the grid cells in our environment can be used to match exactly the grid cells in a routing space. Chapter 4 will introduce this in detail. Figure 3.5 depicts the environment with cells and coordinates.

Figure 3.5: The environment with coordinates

The mines and obstacles in the environment are set in the initialization period of the simulation. Figure 3.6 shows the GUI for our game. With this GUI based game, mines and obstacles can be put into the environment dynamically when the user clicks on the graphic representation of the environment.

Figure 3.6: The GUI of the minesweeping game

In this GUI, once a free solider picks up a mine, it will turn into a busy solider, and he will not pick up any mine until the mine in his hands is released at the border.

3.3 Overall Architecture

There are different approaches to constructing the overall architecture for integrating agents into an HLA simulation. A fundamental concern is to construct a feasible middleware between the agents and the RTI.

One approach is to develop object models for agent federates and for the whole federation when developing agents. In [51], a *KQML-layer* is added to every federate which hosts an agent, in order to let the agents communicate with each other successfully. Other functionality is needed to keep track of remote agents' capabilities in their architecture. The authors concluded that if the HLA/RTI were extended with an agent specific services middleware, it would provide a suitable environment for intelligent agents. In their model, non-agent federates access the RTI directly, while agent federates do so indirectly via the agent middleware.

We initially proposed to implement the soldier federate as a gateway federate, together with using JAVA RMI to realize the simulation of agent behaviors. Figure 3.7 shows the initial proposal of this system.

Later we found this method is complicated for the agent part, and we can realize it in a comparatively easier way. Using the *object-to-agent (O2A) communication channel* provided by the JADE toolkit, the agent code can be directly inserted into the soldier agent federate code, and the pure agent can communicate with the soldier federate using the sensor and effector. Thus we obtained a flexible architecture in our prototype system. Considering the design aspects of the HLA and JADE system, the overall architecture of the current prototype system is presented in Figure 3.8.

Figure 3.9 shows the overall architecture in another perspective, the middleware shown is composed of JADE and the gateway federate. A *gateway federate* is developed to take charge of agents. Agent containers where agents reside are constructed upon it and the *gateway federate* can still access the RTI directly. It is important to note that there can be more than one agent residing in one agent container, and more than one agent container in a JADE platform. However, each agent federate must have its own agent container.

Figure 3.9: Middleware composed of JADE and the gateway federates

In this architecture, the *gateway federate* is in the same JVM (Java Virtual Machine) as the agent and its container. Using the *O2A communication channel*, a gateway federate can communicate with its agent by sending the sensor and effector objects

alternately. This avoids the use of Java RMI which would be necessary if the JADE container and *gateway federate* are executed in different JVMs.

3.4 Object Model - Soldiers and Environment

As mentioned in Chapter 2, the Federation Object Model (FOM) defines types of and relationships among the data exchanged between the federates in a particular federation.

Declaration management in RTI includes publication, subscription, and supporting control functions. Federates that produce object class attributes or interactions must declare exactly what they are able to publish. In the federation, Object classes are comprised of attributes, which describe types of things that can persist. Interaction classes are comprised of parameters. Interaction classes describe types of events. The primary difference between objects and interactions is persistence: Objects persist, interactions do not.

Each federate is responsible for identifying its publication and subscription interests to the RTI using RTIambassador methods `publishObjectClass()` and `subscribeObjectClassAttributes()`.

Two federates are used in this prototype, they are the environment federate and the soldier agent federate, representing the minesweeping environment and the soldier agent respectively. Figure 3.10 depicts the object classes for the minesweeping prototype.

Two main subclasses are defined, namely *Soldier* and *Environment*. The *soldier* represents the agent, and the attribute *Around* is used to express the changed grid cells around the agent after it makes a move, this information has to be passed back to the environment, to update the environment about the picking up of mines.

The use of the *Environment* object class is to make the object management service easier in the federation execution. When the soldier agent federate discovers that the

environment federate exists in the federation, using environment as an object class in the federation is a convenient way to use attributes of the environment itself.

Figure 3.10: Object classes for the minesweeping game

We do not consider *mine* and *obstacle* classes in the FOM in this implementation, as the information about the mines are passed from the environment federate to the soldier federate using interactions instead of objects. But in later implementations, these two classes are considered as object classes of the federation. Figure 3.11 shows the interaction classes in this prototype.

Figure 3.11: Interaction classes for the minesweeping game

Interaction class *CommunicationA* can be used to transfer text messages between the soldier agent and the environment.

The environment federate uses the *CommunicationE* to inform the soldier federate of the detailed information concerning it. This work can also be done using the object publication and subscription, but in order to use both kinds of the declaration management (object update and interactions), we chose to use interactions. Position (*PositionX* and *PositionY*) parameters are used to indicate the current two dimension position (x,y) of the agent in the environment, while *Type* is a string which represents the contents of the nine cells centered on the agent, and *Sign* is used to indicate to the agent if there are obstacles and mines within its sensor region. Figure 3.12 shows one example of the parameters in this class.

Figure 3.12: Illustration of the parameters in the interaction class *CommunicationE*

The extended FED file for our sample implementation is presented in Figure 3.13.

Master of Engineering thesis:
Distributed Simulation of Multi-Agent Systems and their Environments

```
(FED
(Federation minesweeping)
(FEDversion v1.3)
(spaces
)          ;;no routing spaces for the first stage

(objects
(class ObjectRoot
(attribute privilegeToDelete reliable timestamp)

(class MSEntity
(attribute Position_X reliable timestamp)
(attribute Position_Y reliable timestamp)
(attribute Name reliable timestamp)

(class Soldier
(attribute Around reliable timestamp)
)

(class Environment
(attribute Grid reliable timestamp)
)
)
)          ;;end ObjectRoot
)          ;;end objects

(interactions
(class InteractionRoot reliable receive

(class CommunicationA reliable timestamp
(parameter Message)
)

(class CommunicationE reliable timestamp
(parameter Sign)
(parameter Type)
(parameter PositionX)
(parameter PositionY)
)

)          ;;end InteractionRoot
)          ;;end interactions
)          ;;end FED
```

Figure 3.13: The FED file for the minesweeping prototype

The following Table 3.1 shows the object class attribute publications and subscription in our minesweeping prototype implementation.

Table 3.1: Object class attribute publications and subscription in prototype

Object	Federate	
	Environment	Soldier Agent
Soldier privilegeToDeleteObject Position_X Position_Y Name Around	subscribe subscribe subscribe subscribe	default publish publish publish publish publish
Environment privilegeToDeleteObject Position_X Position_Y Grid	default publish	

The soldier agent federate updates the agent's position attributes (*Position_X* and *Position_Y*) once it gets the new positions from the effector. The same applies to the *Around*, which stands for the changed cells information surrounding the old position after the agent makes a move. It is used to change the map of the environment. The *Name* attribute stands for the name of the agent, the environment can use the name to distinguish different agents when there are more than one agent in the environment.

As we discussed above, there is no object publication and subscription relating to the *mine* class, as this can be done using the interaction class described below.

Declaring publication and subscription interest in interaction classes is more straightforward than declaring interest in object classes. Unlike object registration, interactions do not have to be registered because they do not persist. As interactions are "all or nothing", it is impossible to specify interest in particular interaction parameters. Table 3.2 illustrates the interaction class publication and subscription in this prototype system.

Table 3.2: Interaction class publications and subscription in prototype

Interaction	Federate	
	Environment	Soldier Agent
CommunicationA	publish/ subscribe	subscribe/ publish
CommunicationE	publish	subscribe

The environment federate uses the *CommunicationE* to inform the soldier federate of the detailed information concerning it. So the environment federate is responsible for publishing this interaction class, while the soldier agent federate should subscribe to it.

3.5 The Simulation Cycle

Figure 3.14 illustrates the simulation cycle of this game. TAR denotes a Time Advance Request, t is the time step interval and the *lookahead* is also set to t . The logical time corresponding to each TAR is the time each TAR requests.

Figure 3.14: The simulation cycle of the prototype system

When the simulation starts up, the environment federate initializes the obstacles and mines in it, such as the position and the lifetime for the mines as they are set as

dynamic objects. After the initialization period finishes, the environment waits for any information that comes from the soldier agent federate. When the soldier federate code is activated, it will first launch a JADE agent container and initialize a new agent object instance. When the detailed initialization for a new agent is completed in the pure agent code, the soldier federate will resume and initialize its own federate.

In each simulation cycle, when the environment federate gets the reflected attribute values of the soldier agent (step 1), it will do some work on the position to match what is around the agent in the environment (step 2), and then send out that information via an interaction (step 3). The soldier federate receives the interaction (step 4), directly uses the information to construct the sensor (step 5) and then sends both the sensor and effector to the agent using the *object-to-agent communication channel* provided by the JADE toolkit (step 6). We send both sensor and effector references to the agent as the *object-to-agent communication channel* is a one way channel, but information of the agent's actions is passed back to the agent federate via the effector. The soldier federate then will wait until being notified of the agent's new action. For the pure agent side, after it gets the new sensor, it will resume and deliberate its corresponding action using its rules and make a move (step 7), this information will be written to the effector. After this move, the pure agent code will notify the soldier federate and wait again for the next sensor. Once the soldier federate gets the signal from the pure agent, it resumes its work and gets the position of the soldier from the effector (step 8), updates this position to the environment federate (step 9) and then the next simulation cycle will begin.

Figure 3.15 shows the flow charts for the three main processes.

Master of Engineering thesis:
Distributed Simulation of Multi-Agent Systems and their Environments

Figure 3.15: Flow charts of the simulation

3.6 Lessons Learnt

We learned a lot in setting up this preliminary implementation, called Version 1, of the prototype. The system verified our initial proposal to integrate the JADE agent system into an HLA based simulation. The following are the lessons we learnt through setting up this prototype:

- The FED file plays a very important role in the federation execution, where each object class to be used in the federation has to be clearly defined in this file. Moreover, to use each object in the federation, its absolute path has to be specified. For example, the object class *Soldier*, which is the subclass of *MSEntity*, has to be specified as *MSEntity.Soldier* when using the *getObjectClassHandle* method.
- Federation management, declaration management and object management are basic and essential. They are important for the federation execution, and even one small mistake can make the whole system fail.
- Condition variables are used for synchronization between the agent federate and the pure agent.
- There are two time steps for one simulation cycle in this version, one for the agent federate to update its new position to environment federate, the other one for the agent federate to receive the related information via interactions from the environment federate.
- The pure agent can only perceive the world by its sensor and effector, and thus the interface between the pure agent and the federate it resides in is very important. The sensor acts as the agent's eyes and the effector records its latest moves. Information has to be correctly transferred between the pure agent and the sensor/effector; the agent federate and the environment federate.
- The HLA object is not the same as an *object* in object-oriented programming (OOP). An object in OOP is generally defined as an abstraction of a real world entity, it encapsulates the data that describes the characteristics of that entity with the operations that describe the behavior of that entity. But the HLA object is used to establish a basis for information exchange between federation participants. These two *objects* are indeed two totally different concepts.

3.7 Discussion of the Prototype

The success of this preliminary implementation verified the feasibility of the proposed architecture, which integrates agents into HLA-based distributed simulation using the JADE toolkit. But this version has its limitations as a result of the following assumptions we made:

- There is only one agent in the environment.
- One federate is used to host one agent.
- The sensor region of an agent is limited to one grid cell around.
- The environment is a centralized server. When more agents join the federation, this version does not scale well, because the environment federate has to calculate the match for the position of every agent, and send out the corresponding interaction to every agent. The work needs to be divided and distributed among several federates.
- There are two time steps in one simulation cycle, one for the agent federate to update its new information, the other for the agent federate to receive the new information from the environment. This can incur low efficiency of federation execution as the RTI will spend time in synchronization between federates for each time step.

In the following subsections, more details about the prototype will be discussed from the distributed simulation point of view. As defined in the HLA interface specification, there are six services RTI provides to support the federation, so the discussion will be centered on these services, and issues on the whole system will also be included. When more than one agent joins the game, two main problems are how to construct agent sensors using interest management and how to resolve concurrent interactions. These will be investigated in detail in Chapter 4 and Chapter 5 respectively.

3.7.1 Federation management

Besides creating the federation, joining the federation execution and resigning the federation, synchronization points will be required when agent federates need to join the federation at the same simulation time. Synchronization points facilitate

management of agent activities in the simulation, where some problems might happen if those federates do not join the federation simultaneously.

3.7.2 Declaration and Object management

We have shown the FOM for our simple system, which has only one agent in the environment. In future, when more agents join the game and the use of ownership management is needed, a more complete FOM will be used. Mines and Obstacles in the environment of the prototype will be defined as HLA objects in the federation. Chapter 4 will specify the use of those HLA objects.

3.7.3 Ownership management

Ownership management is not necessary when there is only one agent in the game. In version 1, there does exist some kind of ownership of the mines, which are transferred from the environment federate to the soldier agent federate, but it is realized without the use of ownership management.

A mine is alive after it is created, then when a pure agent detects a mine, it will clear the mine from the environment by writing the effector, and this indicates the agent has already picked up a mine. When the soldier federate reads the effector and updates the information to the environment federate, the environment federate will then change the original position of the mine to *null*. So the mine is “transferred” from the environment federate to the agent federate.

However, ownership management will play a very important role in the minesweeping prototype when more agents join the game. Suppose there are two agents that want to pick up the same mine. Because an agent only has a limited sensor region, it will not be aware of the conflicting action.

So from the simulation side, ownership management is a way to solve the problem easier and more efficiently. If one of them must obtain the ownership of that mine before taking any action, the collision will not occur. When the mine is moved by the agent who gets the ownership, the surrounding information for the second agent will

change as the mine is cleared from the original position, so it will not try to pick up the mine.

As the HLA interface specification does not relate time to ownership transfer, it is possible to have unreasonable situations happen. For example, two agent federates may both request the ownership of one object attribute within the same time step, but due to the time delay and other factors, it is possible for the later one to get ownership because the RTI receives it earlier. This problem will be discussed in Chapter 5.

3.7.4 Time management

As we have mentioned above, in each simulation cycle, there are indeed two time steps, one for the agent to update its new positions to the environment, and the other one for the environment do the work and send back information the agent needs. This can be solved by the use of DDM.

The prototype system is a time-stepped simulation now, and the agent will make one action within one time step. As a conservatively synchronized simulation, both the two federates are time-constrained and time-regulating, so their advances of logical time regulate the other and at the same time, are constrained by the other. In the future we may change the time management mechanisms according to the requirement of the agent system. We may develop an event-driven simulation instead of the current time-stepped simulation.

3.7.5 Data distribution management

DDM services control the producer-consumer relationships among federates. In our version 1, the use of the sensor realized the function of interest management: as the sensor has its own region, which is similar to the AOI (area of interest) in DDM, the soldier federate can only receive the information within its region. Although the use of a sensor has the same effect as in DDM, it is not done by the RTI.

But in a more general model, the environment may consist of a number of objects, and changes will be in the form of attribute updates rather than interactions. DDM provides an easy way of filtering this information.

When the simulation scales with more agents and a larger environment, the use of DDM can reduce the communication load of the simulation. Moreover, as each agent may have a different AOI and it may change dynamically, it is easier to use DDM rather than trying to keep track of this in the environment.

Another reason for us to use DDM is the time advancement. In version 1, it takes two time advance requests (TARs) for one cycle of agent behavior: the first TAR for the agent to send its new position to the environment and the second TAR for the agent to receive its sensor from the environment. Using DDM, the agent could specify an AOI and receive the filtered information using one TAR.

Details about how to construct agent sensors using DDM of the HLA will be introduced in Chapter 4.

3.7.6 Sensor and effector

The agent senses the outside world with the sensor and changes the outside world using the effector. So for the agent federate, the interfaces between the federate and the agent are the sensor and the effector. When agents require more information from the environment in which they participate, sensors and effectors are the main changes to be made from the simulation's side as they are the only interface between the agents and the environment.

For each agent in our prototype, the sensor region corresponds to the area of interest of an agent, and is set as the nine grid cells centered on the agent as we presume each agent can only move one grid cell for each time step. Indeed if agents can adopt more than one action and move more than one grid cell, the sensor regions need to be enlarged to get a correct view of the environment in which the agent resides.

Moreover, according to different design requirements, the sensor regions can also vary in shape.

3.7.7 Mutual exclusion

There exist some problems of mutual exclusion. When there is more than one agent in the environment, collisions are unavoidable. Suppose two agents are locally very near in the environment. As they can only perceive the outside world by their sensors, and the sensors have a limited region, it is very possible that they will move to the same cell at the next time step without awareness. The same problem will happen when they try to pick up the same mine. Thus the resolving of mutual exclusion is very important. We have discussed using ownership transfer to solve the problem of picking up the same mine; it is also possible to assign an ownership to each cell. However, there are problems of ownership transfer that have to be solved to ensure the fairness of the game. These problems and solutions will be discussed in detail in Chapter 5.

Based on those considerations, we reinvestigated this system and developed our version 2. Ownership management and data distribution management provided by the RTI are used, which can greatly increase the efficiency of performance and reduce the network load. This version and its further ameliorated versions are discussed in detail in the following two chapters, which introduce how to construct agent sensors using interest management services and how to resolve concurrent interactions in agent based distributed simulations.

CHAPTER

4 CONSTRUCTING AGENT SENSORS WITH INTEREST MANAGEMENT

4.1 Introduction

Using various interest management schemes to construct the sensor of an agent is one of the main research issues of this project. An agent uses its sensor to perceive the environment in which it is embedded. Every soldier agent has a limited knowledge of the environment; they get the information via their own sensors, do deliberations and then act upon the environment using their effectors (see Figure 3.2 in chapter 3).

When modeling agents within federates in the federation, the necessary data transmitted from the environment to the agent federates are the data useful for each sensor of a specific agent. As interest management in large-scale distributed simulations is used to alleviate the network traffic by reducing or eliminating unnecessary data transferred during the simulation, it can be used to meet the demands of this kind of agent-based distributed simulation.

4.2 Implementations

In order to simplify the system, two federates are used in the prototype as described in Chapter 3: the environment federate and the soldier agent federate, representing the minesweeping environment and the soldier agent respectively. Furthermore, currently there is only one soldier agent residing in the JADE container. A more complex system with more agents is investigated in tests described in Chapter 5.

As stated in the previous chapter, the main problem of this prototype is how to construct the sensor of an agent. What an agent can get from the environment wholly depends on the configuration of its sensors. Therefore in the HLA federation, what a soldier agent federate needs is just the information of the nine grid cells centered on its current position instead of all the objects in the environment. This requires mechanisms to filter out all the unnecessary information to reduce the network load and improve the efficiency of the system.

Chapter 3 introduced version 1 of our prototype. This version just uses the object updates and interactions provided by the RTI to send selective information about the environment, it is quite like a request and reply system. In version 2, which is to be described in this chapter, advanced RTI services such as data distribution management (DDM) and ownership management (OM) are used to do data filtering and transfer of ownership. Currently, these versions are both time-stepped simulations, and the agent will make one action within one simulation cycle.

Version 2 treats obstacles and mines as objects in the federation, and it uses ownership transfer for management of dynamic mines. Once a mine has been picked up by a soldier agent, the ownership of that mine will be transferred from the environment federate to the federate in which this agent resides, thus the agent federate is responsible for updating the new position of that mine, or even deleting this mine from the federation. DDM services are used to transfer information about the sensor and reduce the unnecessary data placed on the network. The routing space is statically partitioned into cells. By setting *NumPartitionsPerDimension* in the RID (RTI initialization data) file, we make the cells in the routing space have exactly the

same size as the grid cells of the environment. So the matching between agent sensors and agent DDM regions is much easier. No interactions are used in this version. Figure 4.1 illustrates the object classes for version 2.

Figure 4.1. The object classes of version 2

Based on the previous object classes, in this FOM, Obstacle and Mine classes are added together with their own attributes. As mines are set as dynamic objects in the game, the Life attribute is set to define the lifetime for a mine in the environment. When the time expires, the mine will explode and clear itself from the environment. The ObstacleType attribute of Obstacle class is used to distinguish obstacles like trees, rivers and borders. This is used for the soldier federate to deliberate according to the type of the obstacle, and then it can make an appropriate move.

According the above discussions, the FED file for this version is illustrated in Figure 4.2.

From the figure, we can see that mines and obstacles are set as HLA objects in this version. Also, as DDM services are used in version 2, the definitions for routing spaces have to be specified in the FED file. A routing space is associated with each attribute when using DDM. Moreover, no HLA interactions are used in this version,

so we also do not need to define interactions in the FED file. These are the main differences from the previous FED file of version 1.

```
(FED
  (Federation minesweeper)
  (FEDversion v1.3)
  (spaces
    (space MineField
      (dimension x)
      (dimension y)
    )
  )
  (objects
    (class ObjectRoot
      (attribute privilegeToDelete reliable timestamp MineField)
    )
    (class MSEntity
      (attribute Position_X reliable timestamp MineField)
      (attribute Position_Y reliable timestamp MineField)
      (class Soldier
        (attribute Name reliable timestamp MineField)
      )
      (class Obstacle
        (attribute ObstacleType reliable timestamp MineField)
      )
      (class Mine
        (attribute Life reliable timestamp MineField)
      )
    ) ;;end of MSEntity
  ) ;;end ObjectRoot
) ;;end objects

(interactions
  (class InteractionRoot reliable receive
  ) ;;end InteractionRoot
) ;;end interactions
) ;;end FED
```

Figure 4.2: The FED file for version 2

Using DDM services, the environment federate will update all the objects within it every simulation cycle, while the soldier federate will only discover¹ the objects

¹ Here, *discover* has the general meaning of “being known or visible”, instead of the term *discover* used in the HLA.

within its current subscription region. Table 4.1 depicts the update regions and subscription regions for each federate in this version.

Table 4.1: The DDM regions of version 2

	Soldier Agent Federate	Environment Federate
Subscription Regions	Nine cells centered on agent's position	All cells in environment
Update Regions	Single cell containing Agent	Single cell containing Obstacle/Mine

The simulation cycle of version 2 is shown in Figure 4.3. It is a little different from the first version, so we describe it based on each federate. When the simulation starts up, the environment federate initializes the obstacle objects and mine objects in it, and sets update regions and a subscription region. The ownership management schemes used will be discussed in detail in Chapter 5.

Figure 4.3: The simulation cycle of version 2

In each simulation cycle, the environment federate will see if it receives new positions of agents (E step 1), if so, it will place these agents in its grid cells (E step 2) and unconditionally updates all the objects in the environment (E step 3). Otherwise, the federate will only update all the objects (E step 4). In both conditions, the federate will then wait for the next time step.

For the soldier agent federate, it will launch a JADE agent container and initialize a new agent instance. When the detailed initialization for a new agent is completed in the pure agent code, the soldier federate will resume and initialize its own federate. The soldier federate gets the initial position of the soldier from the effector, sets the subscription/update regions and updates this position to the environment federate.

In each simulation cycle, if there are discovered objects (S step 1), which means these objects are within the subscription region of the soldier, the soldier federate will process this information and construct a sensor (S step 2). Using the object-to-agent communication channel, the federate will send both sensor and effector to the agent (S step 3). When the pure agent gets the new sensor, it will resume and deliberate its corresponding action using its rules and make a move (S step 4).

Furthermore, if the agent decides to pick up a mine, the position of that mine will be also specified. This information will be written to the effector and read by the soldier federate (S step 5). If the federate gets the position of a mine, it will request ownership of that mine from the environment federate (S step 6). After that, the federate will immediately modify the subscription/update regions according to the new position of the soldier agent (S step 7), and update this position (S step 8). However, one problem with this version is that when the next time step begins, the federate will receive outdated objects, which will be discarded (S step 9). Therefore version 2 still has two time steps in one simulation cycle. The reasons for this and possible solutions are explained in detail in the next section.

4.3 Dynamic Regions: Problems and Solutions

We have indicated that although version 2 uses DDM, it still has two time steps in each simulation cycle. This reduces the efficiency of the federation execution as the RTI will spend time in synchronization among federates for every time step. In the description of this game, every time a soldier agent stands on a specific position in the environment, it needs the sensor to do deliberation before moving to another position. Because the soldier agent can move every cycle, its fast changing subscription/update regions make the connectivity established by DDM change dynamically too. This results in the possibility that information provided by the connectivity is outdated for an agent, therefore the agent will make wrong decisions based on the stale information.

For example in version 2 (see Figure 4.3), after deliberation, at a logical time $T+t$, the soldier decides to move from (X_0, Y_0) to position (X_1, Y_1) and immediately changes its subscription/update regions according to this new position. It is supposed to discover the objects around (X_1, Y_1) when it advances its logical time to $T+2t$, so that at the next time step, it can construct a new sensor based on these objects, and the agent can thus deliberate according to the new sensor and make another move. However, unfortunately when the environment federate sends out updates with timestamp $T+2t$, the soldier agent federate has not modified its regions yet. So the connectivity in the communication infrastructure used by the RTI was the one set up based on the

subscription/update regions at logical time T . Thus the soldier agent federate will discover “old” objects around (X_0, Y_0) when it advances its time to $T+2t$. The desired objects will only be discovered at the next time step. So for every action the agent takes, it has to waste an extra time step in the federation to ensure the soldier agent federate will get the right information to construct its sensor and thus move in an appropriate direction.

This problem arises from the fact that DDM services are unsynchronized. Some solutions have been proposed, for example, modifying the RTI to maintain a log of messages as discussed in [57,58]. However, in this solution, the authors use their own developed RTI, and its source code can be revised. For the DMSO RTI we are using, their method is not applicable, thus we need to develop another way to solve the problem.

4.3.1 Enlarged subscription region

After careful investigation, a new algorithm is proposed to solve the dilemma about the above causality problems in this prototype. This algorithm enlarges the subscription region of the soldier agent, thus when the soldier moves to any position at the next time step, this region will still cover the desired objects within the agent’s new sensor region. It is illustrated in Figure 4.4.

Figure 4.4: Enlarged subscription region of the soldier federate

The sensor region is still the nine grid cells centered on the agent. The soldier agent federate only needs to locally filter the objects discovered based on its current sensor region, and discard the redundant information. So even if the DDM connectivity remains unchanged after the soldier federate's modification of its update/subscription regions, we can still make the simulation cycle conform to one time step. A successful version, version 2.1, has been implemented using this algorithm. Its simulation cycle is depicted in Figure 4.5.

Figure 4.5: The simulation cycle of version 2.1

4.3.2 Using advisories

Although the enlarged region algorithm reduces the simulation cycle to one time step, the environment still has to blindly update all the objects in it every time step. This increases the overhead and reduces the scalability of the whole system. It seems ideal if the environment federate can update just the objects the agent needs at each time step. This can be realized using the appropriate methods and advisories provided by DDM, namely *requestClassAttributeValueUpdateWithRegion()* and *enableAttributeScopeAdvisorySwitch()*. After the soldier agent decides to change its position and modify the corresponding update/subscription regions, it will notify the environment federate of this change by

requestClassAttributeValueUpdateWithRegion(). This causes the environment federate to receive the callback *ProvideAttributeValueUpdate()* for the desired objects whose update regions overlap the soldier federate's subscription region. The environment federate will then update only these objects. As the enlarged region algorithm is already used here, the problem of outdated connectivity in the communication infrastructure is avoided. We developed version 2.2 using both the enlarged region algorithm and advisories, this version reduces overheads by sending out data only when necessary. The simulation cycle of this version is shown in Figure 4.6.

Figure 4.6: The simulation cycle of version 2.2

4.3.3 Causality problems of advisories

Advisories do reduce the information transmitted in the federation by sending out just the data needed. However, problems occur occasionally when we conduct experiments for version 2.2. The information each agent federate gets will not always be the latest. After careful investigation, we found it is mainly because all advisories are receive order (RO) messages; however, the results of these services influence the content of subsequent time stamped order (TSO) messages. Those TSO messages are crucial in agent based simulations where agents change their properties and update

this information every time step. Figure 4.7 shows the detailed illustration of this problem while using advisories.

Figure 4.7: Possible error when using advisories

At logical time T , when agent federates get the updated information of agents from the effectors, they will call `requestClassAttributeValueUpdateWithRegion()` to inform the environment federate of their desired object attribute values. This call is a RO message, which means the time its callback `ProvideAttributeValueUpdate()` is received by the environment federate is uncertain. Normally the callback will be received immediately. Ideally, the environment federate will update object attributes at logical time $T+t$ according to the information from the callbacks. Then all agent federates in the federation will get the information they want.

However, different situations will happen when the callback is received at different wall clock times. We suppose:

W_{call} : the wall clock time when the request is sent to the RTI.

$W_{callback}$: the wall clock time when the callback is received.

$W_{update@p}$: the wall clock time when the environment federate updates at logical time p (the update will have a timestamp $p+t$).

The possible situations are:

1. $W_{call} < W_{callback} < W_{update}@T$

If callbacks are received during this period, the environment federate will update information needed for the next time step instead of the current one, because “future messages” are received. To some degree, this is what we want in order to reduce the simulation cycle to only one time step without using enlarged regions. But again, RO messages are uncertain, so we cannot ensure this will always happen. Fortunately, due to the use of enlarged regions, errors will not happen in this situation for version 2.2.

2. $W_{update}@T < W_{callback} < W_{update}@ (T+t)$

This period is safe and it is the normal situation.

3. $W_{callback} > W_{update}@ (T+t)$

If callbacks are received later than the time the environment federate updates at logical time $T+t$ due to network load or other problems, the information each agent federate receives will be out of date. Moreover, it is possible that callbacks of requests at different wall clock times will be received in reverse order. All of these can give rise to errors.

A possible solution would be to encapsulate the RO message into a TSO message, and thus ensure the sequence of information received. However, this method needs an extra time step. So for our system, we use another easier way to solve this problem, that is, instead of letting the RTI transfer requests about data each agent federate wants, we just let the environment federate itself calculate this information based on each agent’s position.

As the environment federate knows the sensor regions and the enlarged region, it can calculate the information each agent federate needs based on the positions of agents in it. All agent federates can get their desired information using Data Distribution Management. Version 2.3 is based on this method. A hashtable and a two dimensional (2D) table are used as data structures for the environment federate. The hashtable is used as a fast way of locating an object based on its object ID, which is assigned by the RTI and exists uniquely throughout the federation. The 2D table is

used to store all the existing obstacles in the environment, including borders, trees, rivers and mines. These obstacles are ordered in the 2D table according to their two dimensional positions (x,y) . Once a mine is created, it will be put into a special hashtable, and then be inserted into this 2D table. Once the environment federate gets reflected soldier information via the RTI, it will search the 2D table according to the position of the soldier, and find obstacles within the sensor region of this soldier. When a mine is picked up and deleted from the federation by a soldier federate, on receiving the callback, the environment federate will search the hashtable of the mine according to the object ID from the callback, get its position, use this position to fetch the mine in the 2D table, and delete it from the table. The use of a hashtable together with a 2D table facilitates both the process of locating objects from their object ID and the process of locating objects according to their two dimensional position (x,y) in the environment. When the system scales and there are more agents, the environment can be partitioned, and each part can also do these calculations.

Figure 4.8 shows the simulation cycle of version 2.3. It is almost the same as version 2.1, but as the environment federate will select information needed (S step 3), it has the same effect as version 2.2 without the causality problems.

Figure 4.8: The simulation cycle of version 2.3

4.5 Experimental Results

Experiments were conducted to evaluate the different approaches for the minesweeping game. The platform for our experiments is three DELL 2.2GHz Pentium 4 computers connected via 100MB Ethernet running Windows XP. JADE agent toolkit version 2.5, DMSO RTI1.3NG-V6 and JAVA jdk1.4.0 are used. In our simulation test, each machine runs a federate. To ensure better results, one computer is used to run the rtiexec and fedexec separately from the federates and all unnecessary outputs were deleted from the source code. The execution time for the agent federate to run 5000 simulation cycles using the four different implementations is recorded. The execution results of the different versions are shown in Figure 4.9.

Figure 4.9: Execution time of different versions

Comparison of the results shows that the proposed enlarged subscription region algorithm, although it increases the network load slightly and introduces some latency in processing the unnecessary information, is still more efficient when the execution time is concerned. Furthermore, the results show version 2.3, which uses both the enlarged subscription region algorithm and the mechanism to avoid causality problems brought by advisories, not only reduces the overhead, but also decreases the federate execution time. Thus it is the most efficient implementation. Our current research is mainly based on version 2.3 described above.

CHAPTER

5 RESOLVING CONCURRENT INTERACTIONS

5.1 Introduction

An *interaction*² can be regarded as the way entities in a system communicate with or influence one another. The results of these interactions normally change behaviors of these entities. In distributed simulations, concurrent interactions can be defined as interactions that happen at the same simulation time or during the same time step. While concurrent interactions are already realized in some CSCW (computer supported collaborative work) systems, very few existing virtual environments permit both distributed and local concurrent interactions between several avatars and one or more objects. One reason can be found in the structure of most existing DVE systems: interactions are implemented by directly manipulating the corresponding object data, which implies an explicit permission to write. In addition, in distributed systems, the problem of message propagation time delays occurs. This has an adverse effect not only on the real-time performance of the system but also on the ability to realize concurrent interactions.

However, as agents have the properties of autonomy, social ability, reactivity and pro-activeness, which offer great convenience in various situations, it is necessary to

² Here *interaction* has a general meaning instead of the term used in the HLA

implement concurrent interactions in agent systems. Thus agents can collaborate to achieve their goals.

5.1.1 Mutually exclusive interactions and collaborative interactions

Concurrent interactions can be either mutually exclusive or collaborative. It depends largely on the intentions of these interactions. Mutually exclusive interactions are interactions that try to access a shared object concurrently, while the shared object only allows a single operation on it at one time. Concurrency control is necessary to ensure inconsistencies do not become unrecoverable [59]. For example, in our prototype system, many agents may try to step to the same grid cell without being aware of the other agents due to limitations of their sensors. However, it is not allowed for a grid cell to have more than one agent in it. So in order to satisfy the rules, only one agent can step to that cell and all other agents need to change their decisions to avoid collisions. On the other hand, there are situations where a shared object permits concurrent accesses to be combined to change properties of the object. These interactions are collaborative interactions. A famous example [60] is that if two different persons lift a table one after another, they are not able to lift it up individually. The table can only be lifted when both of the two people collaborate at the same time, as shown in Figure 5.1.

Figure 5.1: Collaborative concurrent interactions

In our future system, many soldiers may need to collaborate to move a dangerous mine out of the minefield, or to collaborate for even more complicated tasks.

5.1.2 Processing concurrent interactions

According to [60], Distributed Virtual Environments providing concurrent interactions have to deal with two different kinds of problems:

- The concurrent interaction requests have to be detected.
- A “good” mechanism to resolve these requests is necessary.

How different interaction requests are processed, as well as how many participants may interact with an object concurrently, is highly dependent on the object itself. Beside other more specialized methods, there are some alternatives for processing concurrent interactions that are practicable:

- *Priority based interaction request resolving*: In this method, a priority is assigned to each request, and the request with the highest priority will always be processed first while other requests with lower priorities have to be delayed or refused. Normally lower priority requests are refused to ensure the consistency of preceding interactions.
- *Request time dependent interaction sequencing*: This method uses request time at individual sites or at the executing site to determine the order of concurrent interaction requests. This is possible as not even truly concurrent requests will really arrive at the same time at the interaction manager. Sequencing may cause the refusal of some requests, for the same reason as above. The interaction sequence can lead to unexpected results that are different from the intentions of the senders of the requests.

However, the above two methods only cause a sequencing of the requests, and cannot really process concurrent interaction requests. This can be done using the following approaches:

- *Constraint based interaction request resolving*: Using *constraints* is a good approach when a small number of interaction requests are combined with a high degree of freedom. An example based on our game demonstrates this: A

mine object in the environment has three degrees of freedom: two for the position and one for the security type. One user can fix a mine's position by grabbing it. A second user may now operate on the mine to change its security type, but he or she cannot change its position. However, constraints can cause none of the interaction requests be satisfied, if the underlying constraints cannot be resolved.

- *Combining interaction requests*: This method calculates and creates a new request from a series of requests. The calculation depends on the type of interaction and the individual receiving object. The example shown in Figure 5.1 of lifting a table involves combining interaction requests. The table cannot be lifted by any single interaction. Only when two interactions are combined will the outcome of their efforts lift the table up. This may also be used together with priority based request resolving.

5.2 Mutually Exclusive Interactions

5.2.1 Problems of mutually exclusive interactions in the prototype

As far as our prototype is concerned, there are both mutually exclusive interactions and collaborative interactions. With more than one agent participant in the game, problems like stepping to the same empty grid cell or picking up the same mine will occur. This is mainly because of the limited sensor region of each agent as defined by the rules of the game. Figure 5.3 illustrates this problem in our prototype. In this figure, shaded areas are sensor regions for agents. It is clear that agent S_A and agent S_B may step to the same empty grid cell without awareness of the existence of each other.

Figure 5.2: An example of collision

Another example of mutually exclusive interactions in our present system is the problem of picking up the same mine. It is almost the same as that described in Figure 5.2. Soldier agents will try to pick up the same mine as the mine is in their sensor regions, but other agents with the same intentions are just out of their regions. It may be thought that enlarged subscription regions (we use this method for DDM services that are not time stamped) may be one solution, as this allows each agent to get sufficient information of agents around it. However, the intentions of other agents, their future actions (stepping to the empty cell or not, picking up the mine or not) are unknown at that time step. When the next time step begins, the collision may have already happened. Advanced methods have to be adopted to solve these problems. The ownership management services provided by the RTI can be one solution, but this approach has its limitations. A new method using Interaction Resolvers has been developed for agent based distributed simulations. Interaction Resolvers can also be used to solve collaborative interactions, which is part of our future work.

5.2.2 Solution 1: Ownership Management (OM)

Ownership management is used by federates and the RTI to transfer ownership of instance attributes among federates [25]. In our prototype, all attributes of a particular object instance are wholly owned by a single federate. Only one federate can have update responsibility for an individual attribute of an object instance at any given time. This mutually exclusive property of ownership transfer can be used to solve the problems of mutually exclusive interactions.

For example, we can declare mines as HLA objects. Suppose there are two agents that want to pick up the same mine, they must first request the ownership of the mine from the original owner federate of the mine. If one of them must obtain the ownership of that mine instance before taking any action, the collision of concurrent picking up will not occur: when one agent gets the ownership of the mine instance, the other one will be notified of its failure in competition for the ownership. So when the next time step begins, only the winner picks up the mine. The loser will adopt other actions: the surrounding information for it has already changed as the mine is cleared from its original position, so the loser will not try to pick up the mine.

The ownership management methods provide a facility for exchanging attribute ownership among federates in a federation execution using a "push" and/or a "pull" model as described in section 2.3.2. A federate can try to give away responsibility for one or more attributes of an object instance – or *push* ownership [25]. Alternatively, a federate can try to acquire responsibility for one or more attributes of an object instance – or *pull* ownership.

Figure 5.3 shows the ownership model used in our prototype system. *Negotiated push* is initially used (by the environment federate). It is a formal exchange where a federate retains responsibility until a new owner is identified and a formal exchange process is completed. This ensures the attribute update and object deletion responsibilities are not dropped. Also, the *unobtrusive pull* is subsequently used (by the agent federates). So even if the attributes are owned by other federates, this will be reported via the *attributeOwnershipUnavailable()* callback, thus it ensures both winner and loser federates are notified. Otherwise if obtrusive pull is used, the requesting federate will have to keep on waiting for the ownership until the attributes are available, which is not desirable for our prototype.

Figure 5.3: The ownership services used in our prototype system

Each federate that has the ownership of an object can release the ownership whenever it does not need the object. In our implementations, for the conflict of mines, we assure that once the federate gets the ownership of the mine, it can delete it from the federation. For the empty cells, the federate can just release the ownership whenever the winner agent steps out of that cell, using *unconditionalAttributeOwnershipDivestiture()*.

As the HLA interface specification does not relate time stamps to ownership transfer, it is possible to have unexpected situations happen. For example, two agent federates may both request the ownership of one object's attributes within the same time step, but due to the time delay and other factors, it is possible for the later one to get ownership because the RTI receives the request earlier. This receive-order (RO) based priority in getting the ownership can be inconvenient in agent based simulations, where users may prefer to specify that certain agents with defined priority or attributes should win in a competition. Also, if two federates try to acquire and then release ownership of the same object instance within the same time step, it is highly possible that one of them may get the ownership just after the other has released it. Thus the ownership can be acquired twice in one time step, which is illogical.

In the RTI, it makes no sense for each federate to acquire ownership of a specific set of instance-attributes more than once. So when there is more than one agent in one agent federate, these agents will have to compete with each other mainly based on the time they submit their request. Who will obtain the ownership for this federate is unpredictable; as only one agent will “represent” this federate. This is unfair if we want to realize more functions in the competition.

Based on all these considerations, a mechanism using a middleware interaction manager for each federate called *Interaction Resolver* is investigated in our project.

5.2.3 Solution 2: Interaction Resolvers (IRs)

Interaction Resolvers are deliberately designed to resolve the concurrent interactions in our prototype and similar distributed simulations. Each federate will have a local IR, which is used to work out partial collisions that happen within this federate’s subscription region. All the IRs will follow the same working mechanism, thus the result reached in each federate is the same for every shared object. In our game, for each time step, as each agent that resides in the federate only needs to know the objects within its sensor region to make a decision, as long as this information is consistent, accurate and up-to-date, there will be no errors for the decisions of agents in the whole simulation. IRs are distributed in the whole simulation, this greatly reduces the network load compared to the centralized ownership management method. Figure 5.4 shows the overall architecture of the agent based simulation with IRs.

Figure 5.4: The middleware with Interaction Resolvers

In each time step, all interactions³ within a federate's subscription region will be received and stored in a list in its local IR. Then the IR will group them according to the objects and objects' properties of these interactions. For example, in our system, within the interactions received within one time step, some of them may be concerned with stepping to cells, while others may be concerned with picking up mines. The objects of these interactions here are the empty grid cells and the mines. How to determine if these interactions have collisions depends on the definition of the system, or the properties of these objects. In our game, if agents step to grid cells with the same position (x,y) , these interactions are considered as having a collision. Likewise, interactions of picking up mines with the same position (x,y) are also treated as having a collision.

³ Again, here *interaction* has a general meaning instead of the term used in the HLA

Concurrent interactions will be put in different collision lists and then sorted. The sorting of these interactions is mainly based on how the collision is defined. For mutually exclusive interactions, each local IR will compare interactions in each collision list according to the agents' priorities to decide who can be the winner in the collision. If two interactions have the same priority, further properties of each interaction will be used for the decision. These properties must have unique values throughout the whole federation, thus it can quickly bring an end to the winner deciding procedure. We use object instance IDs in our prototype, as these IDs are assigned to each object instance by the RTI, and are unique throughout the federation. So even if two agents have the same priority, they definitely have different instance IDs, and we can then select the winner.

Only the winner can have exclusive privilege to access the shared object, while all losers in this list will require correction. In our game, we roll back losing stepping agents to their previous positions. These corrections are adopted because all agents assume that they have already accessed the shared object and then their owner federate updates this information to other federates. That is why collisions are detected in each relevant IR at the beginning of the next time step. Because IRs are used before constructing sensors that are sent to the corresponding agents, as long as all IRs use the same method of correction, and the results of the correction are consistent, the whole system is free of mistakes as each agent will get corrected status of itself and updated nearby information for its next action. IRs will not function if there are no collisions in one time step. The rollback is executed locally in each federate without further information exchange among federates. Moreover, the rollback state is small, for our game, it is the two dimensional position of each agent. Figure 5.5 illustrates the working of IRs in our system.

```

for each collisionList L[]
    MaxPriority = 0;
    i = 0;
    while i <= L[].length do
        if L[MaxPriority]. priority < L[i]. priority then
            rollback (L[MaxPriority])
            MaxPriority = i
        else if L[MaxPriority]. priority > L[i]. priority then
            rollback (L[i])
        else if L[MaxPriority]. priority == L[i]. priority then
            if L[MaxPriority].mInstanceID > L[i].mInstanceID then
                rollback (L[i])
            else
                rollback (L[MaxPriority]);
                MaxPriority = i
            end if
        end if
        i++
    end while
    winner = L [MaxPriority];
    accept (winner)
end for

```

Figure 5.5: Algorithm of IRs in the prototype system

Admittedly, using IRs does bring temporary inconsistency across a time step in the whole federation, but as far as all the “pure” agents are concerned, in each time step, all IRs immediately correct this inconsistency, and agents will get corrected information and make right decisions. There is no inconsistency in the information received by agents. So in this sense, the whole system is still consistent. Figure 5.6 shows the keeping of consistency in our game. Here TAR means Time Advance Request. In this example, we can clearly see that for federate 1 and federate 2, there

Figure 5.6: How consistency is ensured using IRs in our game

is inconsistency of data among different federates at the same simulation time: each federate may not have the right agent positions for agents of other federates within its subscription region. But as far as all the pure agents are concerned, before they make deliberations in every time step, the use of IRs ensures that all relevant data is consistent.

In our prototype system, in order to resolve mutually exclusive interactions, we also use the enlarged region algorithm implemented in Chapter 4. This is to ensure every agent has sufficient information of possible agents that may have collisions with them. Figure 5.7 shows the use of enlarged subscription regions. Enlarged regions based on the current sensor regions also ensure that all possible agents that may have collisions with a particular agent can be discovered.

Figure 5.7: The use of enlarged regions with IRs

In this example, we can see that soldier agent S_A and S_B cannot perceive each other if they use original subscription regions that equal the size of their sensor regions. So these two agents will have a collision in stepping to the same empty grid cell. However, their local IRs will have no information of each other due to the limitation of the subscription areas. This will definitely cause further errors in the decisions of

each agent and lead to inconsistency of the whole federation. The enlarged region algorithm solves this problem easily. It is used to ensure correct transfer of information between moving agents. Thus it is different from Chapter 4 in which the algorithm is used between agents and the environment. Indeed, as long as each local IR gets sufficient information about each collision, the consistency of the whole system can be assured.

When we change to an event-based distributed simulation, which for certain applications may be a more accurate model to the real system, resolving concurrent interactions is quite different, as there will not be any time-step in the simulation. We may set a period of time as a time threshold, so that two interactions that occur within this threshold are considered concurrent. This is also an issue of our future work in addition to using IRs to solve collaborative concurrent interactions.

In order to compare the efficiency of our interaction resolvers and the ownership management services provided by the RTI, two different test environments are set up. In the first group, all agents are trying to pick up the same mine in a 5*5 minesweeping environment; while in the second group, all agents are stepping into the same grid cell in the environment.

5.3 Comparison of Solutions: Picking Up the Same Mine

In this test, agents distributed in different federates are deliberately set as static agents, that is, they stand still in their initial positions and the only action they can have is to pick up a mine within their sensor regions. The minefield is set as 5*5 grid cells, Figure 5.8 shows this minefield and agents within it. The environment federate will keep on generating mines set in the middle of the minefield once the old mine is picked up and deleted from the federation by its owner agent federate. From the figure we can see, once there is a mine in the minefield, all the agents in that federation will try to pick that mine up as it is within their sensor regions.

Figure 5.8 The minefield for picking up the same mine test

In this test, in order to pick up the mine in the middle of the minefield, either ownership transfer provided by the RTI or interaction resolvers can be used.

5.3.1 Using OM

Mines are set as HLA objects in the federation, once a mine object is initialized, the environment federate automatically has the ownership of that mine and then tries to divest the ownership. All agents interested in picking up this mine should first request ownership of the mine, and only when the federate in which the agent resides acquires the ownership can the agent pick that mine up, and delete it from the federation. Agents failing to pick up the mine will be notified by the RTI and will wait until there is another new mine within their sensor region. Figure 5.9 illustrates the test cycle of this process.

This method will use three time steps in one test cycle. In each test cycle, the environment federate will get reflected agents within the minefield (E step 1), register a new mine in the middle of the minefield and divest the ownership of the mine (E step 2). Once the initialization is finished, the federate will update the mine position (E step 3).

Figure 5.9: The test cycle of test one using OM

For the agent federates, for each test cycle, if there is a discovered mine object within the subscription region of the soldier (S step 1), this information will be written into the sensor. After the construction of the sensors (S step 2) for all the agents within the federate, the federate will send both sensors and effectors to respective agents using the *object-to-agent communication channel* (S step 3). When the pure agent gets its new sensor, it will deliberate its action and decide to pick up a mine if there is a mine in its sensor region (S step 4). This information will be written to the effector and read by the soldier federate (S step 5). If the federate gets the position of a mine, it will request ownership of that mine from the environment federate (S step 6). The agent federate will *tick* and wait for ownership callbacks from the RTI (S step 7). This ensures every federate in the competition knows the result of their ownership acquisition requests. Either *attributeOwnershipAcquisitionNotification* or *attributeOwnershipUnavailable* will be received by agent federates. After this, the agent federate will update agent positions for the agents within it (S step 8). In the

next time step, the federate that gets the ownership will delete the mine from the federation (S step 9), and update agent positions (S step 10).

5.3.2 Using IRs

Interaction resolvers in each federate (including the environment federate) can also be used to resolve the collision problem of picking up the same mine. Thus only the winner of the collision will have the chance to acquire the ownership of the mine, and it will get the ownership without ownership competition in the RTI. This is the main difference from the previous method. Moreover, extra attributes, position of the mine being picked up, are used. This is to provide sufficient information for each local IR in order to decide the collision lists.

For each federate, after it gets reflected objects through the RTI, the local IR will group agents with the same purpose of picking up the mine in the middle, then the IR will determine which agent wins in the competition by its priority. Note that the mechanism to decide who the winner is depends on the application. All other losers in the IR will be notified of their failure in picking up that mine.

The test cycle of using IRs is shown in Figure 5.10.

In each test cycle, the environment behaves almost the same as in the method of using ownership management services. The only difference is the use of the local IR of the environment federate. When the environment federate detects there are more than one agent trying to pick up the same mine, the local IR will resolve this conflict and decide who is the winner (E step 5). Although in the ownership competition, the environment federate does not need to compete with other agent federates, this ensures data consistency in the whole federation.

Figure 5.10: The test cycle of test one using IRs

For each agent federate, if there is a discovered mine object within the subscription region of the soldier (S step 1), this information will be written into the sensors. After the construction of the sensors for all the agents within the federate (S step 2), the federate will send both sensors and effectors to respective agents using the *object-to-agent communication channel* (S step 3). When the pure agent gets its new sensor, it will deliberate its action and decide to pick up a mine if there is a mine in the sensor region (S step 4). This information will be written to the effector and read by the soldier federate (S step 5). The federate will try to pick up this mine (S step 6) and update this information together with agents' positions to the federation (S step 7). Upon receiving this information within its subscription region (S step 8), every agent federate will use its local IR to analyze these data (S step 9). If there are agents trying to pick up the same mine, these agents will be evaluated based on their priorities and

a consistent result will be reached for each IR. Then the agent federate will check to see if any agent in it is the winner in the local IR. If it is, the federate will ask for ownership of the specified mine (S step 10), and tick for callbacks of ownership acquisition (S step 11). When the federate gets the ownership of the mine, it can safely delete it from the federation (S step 12). After this, the federate will update the agent information to other federates in this time step (S step 13).

5.3.3 Experimental results

Timings for different numbers of agents/federates are carried out. The platform for our experiments is five DELL 2.2GHz Pentium 4 computers connected via 100MB Ethernet running Windows XP. JADE agent toolkit version 2.5, DMSO RTI1.3NG-V6 and JAVA jdk1.4.0 are used. In our simulation test, each machine runs a federate. To ensure better results, one computer is used to run the *rtiexec* and *fedexec* separately and all unnecessary outputs were deleted from the source code. The execution time for the agent federate to run 2000 simulation cycles (6000 time steps in this test, test one) using both ownership management and Interaction Resolvers are recorded. For each method, the test is divided into ten different groups, varying in agent and federate numbers. The distribution of agents in varying numbers of federates is carefully designed to get even load balancing and thus achieve the best result. Table 5.1 shows this distribution:

Table 5.1: Distribution of agents in federates

Number of Federates	Total Number of Agents	Distribution
2	2	1+1
	4	2+2
	6	3+3
	8	4+4
3	4	1+1+2
	6	2+2+2
	8	3+3+2
4	4	1+1+1+1
	6	1+1+2+2
	8	2+2+2+2

The experimental results are as follows. Figure 5.11 shows the complete results as the number of federates in the federation increases.

Figure 5.11: Execution time of OM and IRs in picking up the same mine

Figures 5.12 and 5.13 illustrate the difference in the execution time as the number of agents increases for the OM method and IR method respectively. From the figures we can see, as the number of agents increases, the communication time in exchanging position information increases as well.

Figure 5.12: Execution time of OM in picking up the same mine

Figure 5.13: Execution time of IRs in picking up the same mine

Figures 5.14 ~ 5.16 show the comparison of execution time of OM and IRs as the number of agents increase for different numbers of federates.

Figure 5.14: Execution time of OM and IRs for 2 agent federates

Figure 5.15: Execution time of OM and IRs for 3 agent federates

Figure 5.16: Execution time of OM and IRs for 4 agent federates

These results demonstrate that in this test, although ownership transfer is still required after resolving the requests, when IRs are used, IRs are still more efficient than OM. This is because there is only one agent federate asking for ownership in the IR method, while there are more than one agent federate competing for ownership in the OM method, which incurs a higher communication cost.

5.4 Comparison of Solutions: Stepping to the Same Grid Cell

In this test, there is no need to pick up a mine and delete it from the federation. All the agents in the environment just need to step to an empty cell if there is one. The

environment of this test is almost the same as the previous one. The difference is that this time, all agents are mobile, which means they can move into an empty grid cell once there is one within their sensor region. They can also stand still if there are obstacles all around it. We add some obstacles in different test groups in order to test different numbers of agents trying to step to the same grid cell. Once the winner gets the right to step to the empty cell in the middle, it will step out of the cell the next time step, and all other losers will stand still at that time. This is to make the whole test cycle repeat. The minefield is also set as 5*5 grid cells. Figure 5.17 shows an example of this test.

Figure 5.17: The minefield for stepping to the same grid cell test

In this test, both the ownership transfer provided by the RTI and Interaction Resolvers can be used. A federation synchronization point is used in this test in order to start the federation at exactly the same time, otherwise early joined federates will move around before all agents join and this may influence the test results.

5.4.1 Using OM

The specific grid cell in the middle of the minefield is declared as an HLA object in the federation; once the environment federate is initialized, it automatically has the ownership of that grid cell. All agents interested in stepping to this grid cell should first request its ownership, and only when the federate in which the agent resides gets the ownership, can the agent step to the cell. The winner agent will then step out of the cell, and release the ownership of the grid cell for the next possible ownership

competition. Agents failing to step to the grid cell will be notified by the RTI and will stand still until there is another new empty grid cell within their sensor regions. Figure 5.18 illustrates the simulation cycle of this process. Using ownership management services, only the agent federate with ownership of the grid cell will have the right to step to the cell. This is one of the main differences from the method of using interaction resolvers in this test.

Figure 5.18: The test cycle of test two using OM

Unlike test one, there are only two time steps in one test cycle. This is mainly because there is no need to delete the grid cell from the federation, also the environment federate does not need to update the grid cell every time step. As the environment does not have much importance in resolving the ownership conflicts, we just describe the two main types of agent federates: the winner federate (the agent federate who gets ownership of the cell) and the loser federates (other agent federates who are denied ownership). The test cycles for both types are almost the same. They mainly differ in the agent actions after the federate reads the effector information from the pure agent in each time step. The winner agent will step to and step out of

the grid cell, while all loser agents will stand still at their original positions. It is important to note that loser agents can reside in a winner federate. Moreover, the winner federate will have the responsibility to release the ownership in the second time step for the next test cycle.

A test cycle of a winner federate is described as follows. When a test cycle starts, the federate will receive soldier updates within its subscription region (S step 1). It will use this information to construct the sensors for each agent in it (S step 2). The federate will then send both sensors and effectors to respective agents using the *O2A communication channel* (S step 3). When the pure agent gets its new sensor, it will deliberate its action and decide to step to an empty cell within its sensor region (S step 4). The federate will get the decision from effectors (S step 5), and request ownership of the empty grid cell (S step 6). In order to ensure subsequent ownership transfer, the agent federate will *tick* and wait for ownership callbacks from the RTI (S step 7). This ensures every federate in the competition knows the result of its ownership request. The winner agent will step to the cell (S step 8). After this, the agent federate will update agent positions within it (S step 9). In the next time step, the winner federate will first release the ownership of the grid cell instance (S step 10). S step 11 ~ S step 15 are the same as S step 1 ~ S step 5. After the federate has read the effectors, for the winner agent, it will step out of the grid cell (S step 16). Then the federate will update agent positions again (S step 17).

5.4.2 Using IRs

Using IRs, each federate can locally decide who can step to the cell due to the rules of the game. In this test, the one with the highest priority has the privilege. If two agents have the same priority, their unique instance ID in this federation will help in further decision. All losers will be rolled back by the local IR. The same information in each IR ensures the consistency of the view of each agent.

All agents will step to the empty grid cell if there is one within their sensor regions, and only the winner will stay in that cell, while all losers will be rolled back. This is one of the main differences from the method of using OM.

Also, as grid cells need not be HLA objects when using IRs, there is no ownership associated with each grid cell. Thus there is no need for agent federates to request its ownership before they step to the cell, and release ownership when they step out.

Figure 5.19: The test cycle of test two using IRs

As with the OM services method, this method also has two time steps in one test cycle. The test cycle of this method is illustrated in Figure 5.19. The first time steps are the same for both types of agent federates. They all try to step to the empty grid cell. The federate will first get all the soldier information within its subscription region (S step 1). Then it will construct sensors (S step 2) according to the information and send both sensors and effectors to respective agents using the *O2A communication channel* (S step 3). When the pure agents get the new sensors, they will deliberate their actions and write effectors (S step 4). The federate gets each agent's action information from its effector (S step 5), and they know that agents are stepping to the empty grid cell (S step 6). This information is updated by these

federates (S step 7). When the next time step begins, each federate will get relevant updated soldier information (S step 8). All the agents with the same position will be grouped and the local IR will decide who can be the winner (S step 9). Only the winner agent can stay in the position (S step 10), while all losers will be rolled back to their old positions. Because the positions of soldier agents change after IRs are used, sensors and effectors for every agent need to be refreshed to notify the pure agents of their current positions (S step 11). S step 12 ~ S step 15 are the same as S step 2 ~ S step 5. For the winner agent, it will then step back to its original place for the next test cycle (S step 16). Other loser agents will stand still as they have already been rolled back by IRs. At this stage, the environment will become exactly the same as in the initialization, that is, all agents stand in positions around an empty grid cell. The federate will update agents' information before it advances the simulation time (S step 17). Then the next cycle begins.

This method avoids using ownership management in competition for the empty grid cell. This is different from test one where agents try to pick up the same mine. In test one, all agents with the same mine in hand will compete using local IRs, and only the winner can request the ownership of the mine. So in test one, ownership transfer is still used, but in a less competitive way than using the original ownership management services. On the other hand, in test two, the IR method is free from time spent on ownership transfer.

5.4.3 Experimental results

The platform for test two is exactly the same as test one. The execution time for the agent federate to run 2000 simulation cycles (4000 time steps in test two) using both OM and IRs are recorded. This test uses the same distribution of agents in federates as test one. The experimental results are as follows. Figure 5.20 shows the complete results as the number of federates in the federation increases.

Figure 5.20: Execution time of OM and IRs in stepping to the same grid cell

Figures 5.21 and 5.22 illustrate the difference in the execution time as the number of agents increases for the OM method and IR method respectively. From the figures we can see, as the number of agents increases, the communication time in exchanging position information increases as well.

Figure 5.21: Execution time of OM in stepping to the same grid cell

Figure 5.22: Execution time of IRs in stepping to the same grid cell

Figures 5.23 ~ 5.25 show the comparison of execution time of OM and IRs as the number of agents increases for different numbers of federates.

Figure 5.23: Execution time of OM and IRs for 2 agent federates

Figure 5.24: Execution time of OM and IRs for 3 agent federates

Figure 5.25: Execution time of OM and IRs for 4 agent federates

In this test, these results also show that using interaction resolvers are more efficient than using ownership management services provided by the HLA RTI. Besides the communication cost for transfer of position information for both methods, the OM method has the additional cost for resolving ownership transfer.

Compared to test 1, there is no need for the IR method to use ownership management services, so the difference in execution time between the IR method and the OM method in test 2 is larger than that of test 1.

5.5 Summary

From the two tests, we can see clearly that the method of using interaction resolvers is more efficient than the method of using ownership management services provided by the HLA RTI.

Due to the limitation of the environment in our game, there are at most eight agents in a collision. In a more general distributed simulation, where there are many more agents in a collision, the original OM method may not scale well. With the increase of the requests from federates for the same set of instance-attributes, the network load will be greatly increased due to the centralized server. The IR method solves the conflict locally instead. Moreover, there is no bottleneck in the system as all IRs are distributed across the simulation. All these factors reduce the bandwidth requirement in the simulation. Therefore in a wide area network (WAN) or Internet, it is likely that the difference between the IR method and the OM method will be more noticeable. The IR method also avoids all the requesting federates using *tick()* to wait for ownership callbacks from the RTI. As the number of federates increases, if all requesting federates are ticking for the callbacks, it will bring a large synchronization burden for the whole federation.

Moreover, there is a disadvantage of the OM method that all losers in the competition will not know who wins. The way to select the winner is decided by the RTI due to the request time and is unpredictable. This is very inconvenient for agent based simulations where we may want a specific agent with certain advantages to win over others. The IR method enables users to design different winning mechanisms for various implementations. In addition, all the losers in the competition will know the winner, and this is an advantage for practical implementations where further decisions may be based on this knowledge.

We have mentioned that the OM method allows only one agent in each agent federate to request ownership. The IR method definitely does not have this trouble as all agents within each IR are the same regardless of whether or not they belong to the

same federate. It ensures free-competition throughout the federation. Also the rules for the competition can be freely modified by the users.

When we change to an event-based distributed simulation, resolving concurrent interactions is quite different, as there will not be any time steps in the simulation. For the OM method, it is almost impossible as the services are not time stamped. Additional work is needed if OM needs to be used in an event-based distributed simulation. However, for IR we may just need to set a period of time acting as some kind of time threshold.

Based on all these comparisons, we can safely draw the conclusion that the IR method is better than the OM method for solving mutually exclusive interactions in agent based distributed simulations.

CHAPTER

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This project presents a middleware architecture to combine together two modern technologies: agents and Distributed Virtual Environments (DVEs). JADE agents are used in this project to represent certain entities in an HLA based distributed simulation, where fast and accurate decision making is a determining factor of the whole environment. A minesweeping game has been designed and set up to verify the feasibility of our proposed middleware. Two main research issues are carefully investigated based on this game.

An agent perceives the environment in which it is situated via information from its sensor, performs deliberations and acts upon the environment using its effector. Normally, an agent does not need to know all the information of the environment, but just the area in which it is interested. Interest management is used in distributed simulations to alleviate the network traffic by transmitting only necessary data during the simulation. Thus in our system, we compare various interest management schemes to construct the agent sensors. In Chapter 4, different implementations of constructing agent sensors are described in detail. As agents are dynamic in the environment, we propose an enlarged subscription region algorithm to circumvent the causality problem resulting from the lack of synchronization of the HLA services

in logical time. Another causality problem also arises when advisories provided by the HLA are used. We discuss the causes of this problem and propose a mechanism in the environment federate to replace the advisories and avoid the problems they may incur. The experimental results show that our efforts to reduce the simulation execution time are successful.

The other research issue is to resolve concurrent, mutually exclusive interactions in such agent based distributed systems. Resolving concurrent interactions is a key problem of this kind of system, as the shared environment needs to allow agents to interact with the environment in a causally consistent way. There will usually be either mutually exclusive or collaborative interactions. For example, when more than one agent is in the simulation, they may try to step to the same grid cell at the same time, or, as agents are collaborative in nature, they may want to work together to achieve some goals. We successfully developed a middleware called *Interaction Resolvers (IRs)*, which are used to resolve concurrent interactions in our system. IRs can combine the results of concurrent interactions and also at the same time, ensure the consistency of the whole system. Using our prototype system, the minesweeping game, Ownership Management (OM) services provided by the HLA are compared with IRs. Two different situations of mutually exclusive interactions: stepping to the same grid cell and picking up the same mine are investigated using these two solutions. Besides the flexibility and convenience IRs provide, experimental results also show that the distributed IRs save more execution time than the centralized OM.

6.2 Future Work

By the end of this project, we have successfully integrated the JADE agent platform into HLA based distributed simulations, and investigated various interest management methods to construct agent sensors. In order to resolve concurrent interactions in agent based systems, we also developed Interaction Resolvers as a middleware component. The problems of unsynchronized HLA services in agent based distributed simulations are solved by our algorithms. Future work of this project should be centered on the following three aspects:

- *A more general IR*: A more complete IR component needs to be devised to satisfy the requirements of collaborating agents. Current IRs in our system can only deal with mutually exclusive interactions. When a collision happens, the local IR in each federate can select a winner due to certain properties of agents, or user defined rules. When collaborative concurrent interactions happen, we need to set up certain rules for local IRs to classify different interactions according to certain properties, and act upon each of them to achieve a combined interaction. This is not difficult based on what we have achieved so far. Moreover, we need to generalize the current IRs algorithm to make it applicable to most agent based distributed simulations, or distributed simulations where concurrent interactions happen. For example, IRs can analyze the common features of interactions and divide them into different categories [61].
- *A general way to synchronize DDM services*: There are two types of services in the HLA: time managed (those resulting in the reception of time stamp order messages) and not time managed (resulting in receive order messages). Many services in the HLA such as DDM are non-time managed. When agents join the federation, their highly mobile features require very precise time management to ensure system consistency and causality. So once unsynchronized services are used, it is highly possible that causality problems may arise in this kind of system. Our current way of solving these problems in Chapter 4 is efficient in our game or similar systems. However, they may need to be revised for use in other systems, for example, if the assumptions of the current game are relaxed. A more general way to solve these problems can be devised by exploring a software layer between the RTI and federates. In this layer, a possible approach is that the relevant unsynchronized RTI services can be encapsulated with time stamps and sent out using existing time stamped RTI services. This will ensure the synchronization of all the services relevant to agents and guarantee the precision of the whole system.
- *Event-based distributed simulation*: Currently we are using time-stepped simulation. In the future, event-based simulation may be more appropriate for our system, as agents are mobile in the environment, and not all of them will

have actions within each time step. Switching from time-stepped simulation to event-based simulation may be a more natural approach for agent based simulations, especially when more agent activities are involved. When we go from a time-stepped simulation to an event-driven simulation, an agent does not need to receive sensor information so frequently. An agent only needs sensor information when it changes its current position or the environment information around it changes, then its federate can advance the simulation time to the time of this event. When the agent does not move and the information around it remains unchanged, there is no need to advance time.

6.3 Summary

In this project, we verify the possibility of integrating JADE agents into HLA based distributed simulation to represent certain entities. A prototype system named *minesweeping game* is proposed and developed. Two research issues, interest management and resolving mutually exclusive interactions, are investigated carefully. We compare different interest management schemes in constructing agent sensors, and propose algorithms to circumvent causality problems resulting from unsynchronized HLA services. Moreover, we successfully developed a middleware layer between agents and federates called *Interaction Resolvers* to resolve concurrent, mutually exclusive interactions. Experimental results prove that the IRs are more efficient than ownership management services provided by the HLA in resolving mutually exclusive interactions in our system.

APPENDIX A SYSTEM CONFIGURATION

To set the system environment for the HLA, JADE and Protégé (an Ontology tool) [62]:

1. Install the RTI V6 on the machine, as well as the java binding, and put them under the same folder. Also copy all JADE files into a folder called JADE:

2. Set the system environment variables for both JADE and HLA:

Master of Engineering thesis:
Distributed Simulation of Multi-Agent Systems and their Environments

CLASSPATH

```
.,;%JAVA_BINDING_HOME%\javalib;D:\mc\jade\lib\jade.jar;D:\mc\jade\lib\jadeTo  
ols.jar;D:\mc\jade\lib\Base64.jar;D:\mc\jade\lib\iiop.jar;D:\mc\jade\classes;C:\Proteg  
e-2000 1.7 \protege.jar;C:\Protege-2000 1.7 \plugins\query_tab.jar;C:\Protege-2000  
1.7\plugins\rdf_backend.jar;C:\Protege-2000 1.7 \plugins\rdf-api.jar;C:\Protege-2000  
1.7\plugins\standard_extensions.jar;C:\Protege-2000  
1.7\plugins\xerces.jar;C:\Protege-2000 1.7\plugins\beangenerator.jar;C:\Protege-2000  
1.7\plugins\jess.jar;C:\Protege-2000 1.7\plugins\JessTab10.jar
```

Path

```
%RTI_HOME%\%RTI_BUILD_TYPE%\bin
```

JAVA_BINDING_HOME

```
%RTI_HOME%\%RTI_BUILD_TYPE%\apps\JavaBinding
```

RTI_HOME

```
C:\dms0\RTI1.3NG-V6
```

RTI_BUILD_TYPE

```
Win2000-VC6
```

3. Make a batch file for the application. For example:

RTI:

```
C:\DMSO\RTI1.3NG-V6\Win2000-VC6\bin\rtiexec -multicastDiscoveryEndpoint  
224.9.9.9:12347  
pause
```

Rune(Environment):

```
java -cp .;D:\mc\rti EnvironmentFed 10 10 300  
pause
```

Runa(Agent):

```
java SoldierAgent soldieragent_1 8 8 170  
pause
```

4. Add the system environment variables for Protégé-2000 1.7. (It can also copy to directory other than C:\...) It is important to ensure MSGontology in the same directory of the program files.

APPENDIX B PUBLICATIONS

F. Wang, S.J. Turner and **L. Wang**, “*Integrating Agents into HLA-Based Distributed Virtual Environments*”, Proceedings of 4th Workshop on Agent-Based Simulation, Montpellier, France, April 2003, pp. 9-14.

L. Wang, S.J. Turner, F. Wang, “*Interest Management in Agent-Based Distributed Simulations*”, Proceedings of 7th IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT 2003), Delft, Netherlands, October 2003, pp. 20-27.

F. Wang, S.J. Turner and **L. Wang**, “*Agent Communication in Distributed Simulations*”, Proceedings of 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS’04), New York, US, 2004, Workshop W19, pp. 11-20.

L. Wang, S.J. Turner and F. Wang, “*Resolving Mutually Exclusive Interactions in Agent Based Distributed Simulations*”, Proceedings of Winter Simulation Conference 2004 (WSC’04), Washington DC, US, 2004, pp. 783-791.

F. Wang, S.J. Turner and **L. Wang**, “*Multi-Agent Interactions in Distributed Virtual Worlds*”, Proceedings of IEEE TENCON 2004, Chiang Mai, Thailand, 2004, Volume B, pp 345-348.

REFERENCES

- [1] M. Wooldridge, “*Agent-based Software Engineering*”, IEE Proceedings on Software Engineering, 144(1), February 1997, pp. 26-37.
- [2] A.M. Uhrmacher, P.A. Fishwick and B.P. Zeigler (eds), “*Special Issue on Agents in Modeling and Simulation: Exploiting the Metaphor*”, Proceedings of the IEEE, 89 (2), February 2001, pp. 127-213.
- [3] S. Singhal and M. Zyda. *Networked Virtual Environments: Design and implementation*, ACM Press, ISBN 0-201-32557-8, 1998.
- [4] Defense Modeling and Simulation Office, *High Level Architecture RTI Interface Specification, Version 1.3*, 1998.
- [5] JADE agent toolkit. Available at <http://sharon.cselt.it/projects/jade>.
- [6] F. Bellifemine, A. Poggi, and G. Rimassa, “*JADE - A FIPA-compliant agent framework*”, Proceedings of PAAM'99, April 1999, pp. 97-108.
- [7] FIPA Agent Management Specification Technical Report SC00023J. Available at <http://www.fipa.org/>. Dec 2002.
- [8] R.M. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley Interscience, January 2000.
- [9] A. Pope, BBN Report no. 7102, “*The SIMNET Network and Protocols*”, BBN Systems and Technologies, Cambridge, Massachusetts, July 1989.

- [10] IEEE, International Standard, ANSI/IEEE Std 1278-1993, *Standard for Information Technology, Protocols for Distributed Interactive Simulation*, March 1993.
- [11] M. Macedonia, M. Zyda, D. Pratt, and P. Barham, “*Exploiting Reality with Multicast Groups: a Network Architecture for Large Scale Virtual Environments*”, Virtual Reality Annual International Symposium, March 1995, pp. 2–10.
- [12] M.R. Macedonia, M. J. Zyda, D.R. Pratt, P.T. Barham, and S. Zeswitz, “*NPSNET: A Network Software Architecture for Large Scale Virtual Environments*”, In Presence, vol. 3, Fall 1994, pp 265-287.
- [13] T. Chiueh, A. Ballman, and P. Pradhan, “*Distributed Systems Support for Networked Games*”, Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, 1997, pp. 99-104.
- [14] T.A. Funkhouser, “*RING: A Client-Server System for Multi-User Virtual Environments*”, Computer Graphics (1995 SIGGRAPH Symp. Interactive 3D Graphics), Monterey, CA, April 1995, pp. 85-92.
- [15] M.R. Garey and D.S. Johnson, “*Computers and Intractability: a Guide to the Theory of NP-Completeness*”, W.H. Freeman and Company, 1978.
- [16] G. Karypis, V. Kumar, “*Multilevel Scheme for Partitioning for Irregular Graphs*”, Technical Report, Department of Computer Science, University of Minnesota, 1997.
- [17] K.Y. So Oldfield, “*Partitioning Problem in a Very Large Scale Distributed Virtual Environment*”, M. Phil second term paper, Technical report RM1026-TR98-0415, Chinese University of Hong Kong, 1998.

- [18] Peter T.S. Tam, “*Communication Cost Optimization and Analysis in Distributed Virtual Environment*”, M. Phil second term paper, Technical report RM1026-TR98-0412, Chinese University of Hong Kong, 1998.
- [19] J. Lui, O. So, M. Chan, and T. Tam, “*Dynamic Partitioning for a Distributed Virtual Environment*”, Proceedings of the 3rd High Performance Computing Asia Conference (HPC Asia'98), 1998.
- [20] J. S. Dahmann, F. Kuhl, R. Weatherly, “*Standards for Simulation: As Simple as Possible but not Simpler - The High Level Architecture for Simulation*”, Simulation 71(6), 1998, pp. 378-387.
- [21] F. Kuhl,, R. Weatherly and J. Dahmann, *Creating Computer Simulation Systems: An Introduction to High Level Architecture*, Prentice Hall, 1999.
- [22] IEEE P 1516, *Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -Framework and Rules*, September 1998.
- [23] IEEE P 1516.1, *Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -Federate Interface Specification*, April 1998.
- [24] IEEE P 1516.2, *Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -HLA Object Model Template (OMT)*, February 1998.
- [25] US Department of Defense, *High Level Architecture RTI-NG 1.3v5 Programmer's Guide*, February 2002.
- [26] F. Moradi, R. Ayani, and G. Tan, “*Some Ownership Management Issues in Distributed Simulations using HLA/RTI*”, Parallel and Distributed Computing Practices, 4(1), June 2001, Nova Science Publishers, pp. 77-91.

[27] G. Sauerborn, G. Tan, G. Moss, P. Oxenberg, F. Moradi, R. Ayani, “*HLA Ownership Management Services: We Almost Got it Right*”, Proceedings of Simulation Interoperability Workshop, Fall 2000, Orlando, Florida, September 2000, 00F-SIW-076.

[28] K.L. Morse, “*Interest Management in Large Scale Distributed Simulations*”, Tech. Rep. 96-27, Department of Information and Computer Science, University of California, Irvine, 1996.

[29] J. Smith, K. Russo, and L. Schuette, “*Prototype Multicast IP implementation in ModSAF*”, Proceedings of the twelfth Workshop on Standards for the Interoperability of Distributed Simulations, 1995, pp. 175–178.

[30] T.W. Mastaglio and R. Callahan, “*A Large-Scale Complex Virtual Environment for Team Training*”, IEEE Computer, 28(7), July 1995, pp. 49–56.

[31] J.O. Calvin, C.J. Chiang, and D.J. Van Hook, “*Data subscription*”, Proceedings of the Twelfth Workshop on Standards for the Interoperability of Distributed Simulations, March 1995, pp. 807–813.

[32] J.S. Steinman and F. Weiland, “*Parallel Proximity Detection and the Distribution List Algorithm*”, Proceedings of the 1994 Workshop on Parallel and Distributed Simulation, July 1994, pp. 3–11.

[33] D.J. Van Hook, J. Calvin, M. Newton, and D. Fusco, “*An Approach to DIS Scalability*”, Proceedings of the 11th Workshop on Standards for the Interoperability of Distributed Simulations, 1994, pp. 347–356.

[34] A. Berrached, M. Beheshti, O. Sirisaengtaksin, and A. Korvin, “*Alternative Approaches to Multicast Group Allocation in HLA Data Distribution*”, Proceedings of the 1998 Spring Simulation Interoperability Workshop, 1998, 98S-SIW-184.

[35] J.O. Calvin and D.J. Van Hook, “*Agents: An Architectural Construct to Support Distributed Simulation*”, Proceedings of the 11th Workshop on Standards for the Interoperability of Distributed Simulation, September 1994, pp. 357-366.

[36] K.L. Morse, L. Bic, M. Dillencourt, and K. Tsai, “*Multicast Grouping for Dynamic Data Distribution Management*”, Proceedings of the 31st Society for Computer Simulation Conference (SCSC '99), 1999.

[37] J. Saville, “*Interest Management: Dynamic Group Multicasting using Mobile JAVA Policies*”, Proceedings of the 1997 Fall Simulation Interoperability Workshop, 1997, 97F-SIW-020.

[38] S.B. Hall, B.P. Zeigler, and H. Sarjoughian, “*Joint MEASURE: Distributed Simulation Issues in a Mission Effectiveness Analytic Simulator*”, Proceedings of the Simulation Interoperability Workshop, Orlando, FL, 1999, 99F-SIW-159.

[39] S.B. Hall, “*Using Joint MEASURE to study Tradeoffs between Network Traffic Reduction and Fidelity of HLA compliant Pursuer/Evader Simulations*”, Proceedings of the Summer Simulation Conference, Vancouver, Canada, 2000, Society for Computer Simulation, pp. 287-294.

[40] H.S. Sarjoughian, B.P. Zeigler, and S.B. Hall, “*A Layered Modeling and Simulation Architecture for Agent-Based System Development*”, Proceedings of the IEEE, 89(2), 2001, pp. 201-213.

[41] R.K. Blashfield, M.S. Aldenderfer, and L.C. Morey, “*Cluster Analysis Software*”, In Handbook of Statistics vol. 2, North-Holland Publishing Company, Amsterdam, 1982.

[42] J. Yu, C. Raczy, and G. Tan, “*Evaluation of a Sort-based Matching Algorithm for DDM*”, Proceedings of 16th Workshop on Parallel and Distributed Simulation, Washington D.C., USA, May 2002, pp. 68-75.

[43] R. Ayani, F. Moradi and G. Tan, "*Optimising Cell-size in Grid-Based DDM*", Proceedings of 14th Workshop on Parallel and Distributed Simulation, Bologna, Italy, May 2000, pp. 93-100.

[44] D.J. Van Hook, S.J. Rak, J.O. Calvin, "*Approaches to RTI Implementation of HLA Data Distribution Management Services*", Proceedings of the 15th Workshop on Standards for the Interoperability of Distributed Simulations, September 1996, pp. 535-544

[45] G. Tan, Y.S. Zhang and R. Ayani, "*A Hybrid Approach to DDM*", the 4th International Workshop on Distributed Simulation and Real-Time Applications, San Francisco, USA, August 2000, pp. 55-61.

[46] A.M. Uhrmacher and K. Gugler, "*Distributed, Parallel Simulation of Multiple, Deliberative Agents*", Proceedings of the 14th Parallel and Distributed Simulation Conference (PADS'2000), May 2000, pp. 101-110.

[47] G. Theodoropoulos and B. Logan, "*A Framework for the Distributed Simulation of Agent-Based Systems*", Proceedings of the 13th European Simulation Multiconference (ESM'99), Helena Szczerbicka, Ed. SCS, June 1999, vol. 1, pp. 58-65.

[48] M. Lees, B. Logan and G. Theodoropoulos, "*Simulating Agent Based Systems with HLA -- The Case of SIM_AGENT*", Proceedings of the 2002 European Simulation Interoperability Workshop (ESIW'02), June 2002, pp. 285-293.

[49] G. Tan and L. Xu, "*An Agent-based DDM Filtering Mechanism for HLA*", Special Issue on Software Agents and Simulation, Simulation, 76(6), June 2001, pp. 329-344.

[50] G.J. Valentino, K. Morse, T. Kniola and S. Khalil, “*An Agents Toolkit to Support Distributed Simulations*”, Proceedings of Simulation Interoperability Workshop, Orlando, Florida, Spring 1998, 98S-SIW-009.

[51] J. Andersson and S. Löf. “*HLA as Conceptual Basis for a Multi-Agent Environment*”, Technical Report 8TH-CGF-033, Pitch Kunsapsutveckling AB, 1999.

[52] R. Minson, G. Theodoropoulos, “*Distributing RePast Agent-Based Simulations with HLA*”, Proceedings of the 2004 European Simulation Interoperability Workshop, Edinburgh, 2004, 04E-SIW-046.

[53] M. Adelantado, “*Airport Simulation using the High Level Architecture*”, 2001 European Simulation Interoperability Workshop, June 25-27, University of Westminster, Harrow Campus, Middlesex, UK, 2001, 01E-SIW-005.

[54] F. Chantemargue and B. Hirsbrunner, “*A Collective Robotics Application Based on Emergence and Self-Organization*”, Technical report, PAI group, Fribourg, Switzerland, 1999.

[55] M.E. Pollack, M. Ringuette, “*Introducing the Tileworld: Experimentally evaluating agent architectures*”, National Conference on Artificial Intelligence, 1990, pp. 183-189.

[56] X. Shen, R. Hage and N.D. Georganas, “*Agent-aided Collaborative Virtual Environments over HLA/RTT*”, Proceeding of 3rd International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RT'99), Greenbelt MD, Oct. 1999, pp. 128-136.

[57] R.M. Fujimoto, “*Time Management of Unsynchronized HLA Services*”, 1999 Fall Simulation Interoperability Workshop, 1999, 99F-SIW-165.

Master of Engineering thesis:
Distributed Simulation of Multi-Agent Systems and their Environments

[58] I. Tadic and R.M. Fujimoto, “*Synchronized Data Distribution Management in Distributed Simulations*”, Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS '98), May 1998, pp. 108-115.

[59] G. Lann, “*Consistency, Synchronization and Concurrency Control*”, Distributed Data Bases, Cambridge University Press, USA, 1980, pp. 195-221.

[60] W. Broll, “*Interacting in Distributed Collaborative Virtual Environments*”. Proceedings of the IEEE VRAIS'95 Conference, IEEE Computer Society Press, Las Alamitos, California, March 1995, pp. 148-155.

[61] A. Natrajan, P.F. Reynolds, Jr. “*Resolving Concurrent Interactions*”, Proceedings of 3rd International Workshop on Distributed Interactive Simulation and Real Time Applications, October 1999, pp. 85-92.

[62] Protégé Project. Protégé tools, available at <http://protege.stanford.edu/>.

Master of Engineering thesis:
Distributed Simulation of Multi-Agent Systems and their Environments